Spring 2018

Unsupervised Parkinson's Disease Assessment

Alexander S. Adranly
Santa Clara University, aadranly@scu.edu

Senbao Lu
Santa Clara University, slu1@scu.edu

Yousef Zoumot
Santa Clara University, yzoumot@scu.edu

SANTA CLARA UNIVERSITY

Department of Computer Science Engineering
Department of Bioengineering

I HEREBY RECOMMEND THAT THE THESIS PREPARED
UNDER MY SUPERVISION BY

Alexander S. Adranly, Senbao Lu, Yousef Zoumot

ENTITLED

Unsupervised Parkinson's Disease Assessment

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREES OF

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE ENGINEERING
AND
BACHELOR OF SCIENCE
IN
BIOENGINEERING

Thesis Advisor	6/13/18 date
Thesis Advisor	6/13/18 date
Department Chair	6/13/18 date
Department Chair	6/13/18 date

Unsupervised Parkinson's Disease Assessment

by

Alexander S. Adranly
Senbao Lu
Yousef Zoumot

SENIOR DESIGN PROJECT REPORT

Submitted to
the Department of Computer Science and Engineering
and
the Department of Bioengineering

of

SANTA CLARA UNIVERSITY

in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Computer Engineering
Bachelor of Science in Bioengineering

Santa Clara, California

Spring 2018

Unsupervised Parkinson's Disease Assessment

Alexander S. Adranly
Senbao Lu
Yousef Zoumot

Department of Computer Engineering
Department of Bioengineering
Santa Clara University
June 14, 2018

ABSTRACT

Parkinson's Disease (PD) is a progressive neurological disease that affects 6.2 million people worldwide. The most popular clinical method to measure PD tremor severity is a standardized test called the Unified Parkinson's Disease Rating Scale (UPDRS), which is performed subjectively by a medical professional. Due to infrequent checkups and human error introduced into the process, treatment is not optimally adjusted for PD patients. According to a recent review there are two devices recommended to objectively quantify PD symptom severity. Both devices record a patient's tremors using inertial measurement units (IMUs). One is not currently available for over the counter purchases, as they are currently undergoing clinical trials. It has also been used in studies to evaluate to UPDRS scoring in home environments using an Android application to drive the tests. The other is an accessible product used by researchers to design home monitoring systems for PD tremors at home. Unfortunately, this product includes only the sensor and requires technical expertise and resources to set up the system. In this paper, we propose a low-cost and energy-efficient hybrid system that monitors a patient's daily actions to quantify hand and finger tremors based on relevant UPDRS tests using IMUs and surface Electromyography (sEMG). This device can operate in a home or hospital environment and reduces the cost of evaluating UPDRS scores from both patient and the clinician's perspectives. The system consists of a wearable device that collects data and wirelessly communicates with a local server that performs data analysis. The system does not require any choreographed actions so that there is no need for the user to follow any unwieldy peripheral. In order to avoid frequent battery replacement, we employ a very low-power wireless technology and optimize the software for energy efficiency. Each collected signal is filtered for motion classification, where the system determines what analysis methods best fit with each period of signals. The corresponding UPDRS algorithms are then used to analyze the signals and give a score to the patient. We explore six different machine learning algorithms to classify a patient's actions into appropriate UPDRS tests. To verify the platform's usability, we conducted several tests. We measured the accuracy of our main sensors by comparing them with a medically approved industry device. The our device and the industry device show similarities in measurements with errors acceptable for the large difference in cost. We tested the lifetime of the device to be 15.16 hours minimum assuming the device is constantly on. Our filters work reliably, demonstrating a high level of similarity to the expected data. Finally, the device is run through and end-to-end sequence, where we demonstrate that the platform can collect data and produce a score estimate for the medical professionals.

Acknowledgements

This research was supported by Engineering Undergraduate Programs Senior Design Project Funding at Santa Clara University. We would like to thank Prof. Dezfouli and Prof. Yan for their earnest and tireless inculcation. We would also like to give our thanks to Dr. Jayant Menon and Mr. David Haygood for their insight of Parkinson's Disease, which inspired us to pursue this project. Finally, we would like to thank our families for their supporting and understanding. We would also like to acknowledge the amazing develops, whose open source APIs we are using. Without their contributions it would have been significantly harder to design our system within our given time frame.

Contents

List of Figures

List of Tables

Glossary of Terms

1. General

1. MCU: Microcontroller
2. UPDRS: Universal Parkinson's Disease Rating Scale
3. UPDA: Unsupervised Parkinson's Disease Assessment
4. QoS: Quality of Service
5. Instance Data: A single set of data collected from the wearable device
6. Sample Data: A group of instance data within a time period

2. Biological

1. PD: Parkinson's Disease
2. DBS: Deep Brain Stimulation
3. Dorsum of Palm: The back side of a hand.
4. Distal Phalanx: The bone at the distal of each fingers
5. Metacarpophalangeal (MCP) Joints: The joint at the base of the finger
6. Flexor Digitorum Superficialis Muscle: An extrinsic flexor muscle of the fingers at the proximal interphalangeal joints. The primary function is flexion of the middle phalanges of the fingers at the proximal interphalangeal joints.

3. Computational

1. CPU: Central Processing Unit
2. ADC: Analog To Digital Converter
3. IMU: Inertial Measurement Unit
4. sEMG: Surface Electromyography
5. MEMS: Microelectricalmechanical System
6. Accelerometer: MEMS that measure acceleration
7. Gyroscope: MEMS that measures rotational velocity
8. Magnetometer: MEMS that measures orientation relative to the earth's magnetic field

9. I2C: Inter-Integrated Circuit
10. Bit: The smallest packet of information in a software system, which can be either true or false.
11. Byte: Eight bits

Chapter 1

Introduction

1.1 Motivation

Parkinson's Disease (PD) is a progressive neurological disease that affects 6.2 million people around the world [1]. Many advanced PD patients suffer from severe motor symptoms including arm and hand tremors, which increases the difficulty of performing daily tasks. Additionally, PD patients can suffer from of other non-motor symptoms [2] varying from depression to sleep disorders.

Currently, Parkinsonian tremors can be effectively controlled through oral-medical therapies such as Levodopa, or invasive medical devices such as Deep Brain Stimulation (DBS) [3]. Although these treatments work, they are costly (approximately $\$22,800$ per patient [4]), highly patient selective, and can have many high risks [2, 3]. Furthermore, as the disease progresses, both treatments need to be adjusted according to the patient's symptoms to optimize their performance [5]. Therefore, it is important to continuously monitor a patient's symptoms for better treatment results.

Clinically, the severity of Parkinsons disease is measured through a series of tests called the Unified Parkinson's Disease Rating Scale (UPDRS), where the clinician will subjectively evaluate the patient through a series of questions and tasks. Since measuring the severity of this disease is important for understanding and applying treatment, there has been a push to automate the quantification of the UPDRS for more unbiased results.

According to a recent review there are two devices recommended to objectively quantify PD symptom severity [6]. Both devices record a patient's tremors using inertial measurement units (IMUs). One is not currently available for over the counter purchases, as they are currently undergoing clinical trials. It has also been used in studies to evaluate to UPDRS scoring in home environments using an Android application to drive the tests. The other is an accessible product used by researchers to design home monitoring systems for PD tremors at home. Unfortunately, this product includes only the sensor and requires technical expertise and resources to set up the system to perform UPDRS.

1.2 Problem

For home use and evaluating Parkinson's Disease based on the UPDRS, these solutions fall short. Most devices still do evaluate Parkinson's Disease in a controlled environment, but many researchers and health care professionals are pushing to automate this process [7, 8, 9, 10]. To monitor PD for medication adjustment, a patient must continuously perform these choreographed exercises, which places a responsibility on the user that one might consider unreliable.

1.3 Contributions

In this project, we create a device that will utilize patients daily actions instead of choreographed actions to monitor and quantify Parkinson's Disease symptoms based on part of the motor examination section of the UPDRS. Our solution focuses on the monitoring and scoring of tremors using a patients normal movements throughout their daily life. The solution consists of a wearable device wirelessly connected to a gateway. The wearable device monitors the patient with sufficient precision and sends the information to the gateway on which it can perform a more thorough analysis and define tremors. With our solution, researchers will be able to better understand the patterns and causes of PD tremors which could help patients continue living their daily lives with confidence. Furthermore, this can ultimately be extended to monitoring and evaluating all other tremors.

Chapter 2

System Overview

2.1 System Architecture

Looking at figure 2.1, there are several different elements that make up the entire system. At the heart of the system are elements **1** and **2**, which both reside in the home of the patient. Element **1** is a wearable device that the patient wears on their arm for a duration of the day. This device is responsible for collecting the raw data, performing some preprocessing, and transferring the information to element **2**. Element **2** is a networking device that collects the raw information from the wearable device and performs more intense analyses on the code; producing the UPDRS scores for the user's condition. Element **3** acts as a data storage system that will hold all of the patient information that the wearable collects as well as the analysis of the raw data. Element **5** is a personal computer where the appropriate amount of information can be viewed by the client. For instance, a personal computer at the patients home would be able to view the UPDRS scores while a physician's computer would be able to view the raw data as well as the automated UPDRS scores. However, in order to promote simplicity for the patient, it is optional for the patient to have a computer. Element **4** is the Internet infrastructure, which is used to transfer patient data from the patient's home.

For the current scope of the project, we are currently focusing on the design and development of elements **1**, **2**, and **3** as shown in figure 2.1.

The architecture for this system holds both elements of both a data-centric architecture as well as a client-server architecture. The wearable device and the networking device together form a type of client-server architecture because both sides will be doing a considerable amount of processing. The personal computer clients that interact with the networking device behaves more as a data-centric architecture, since the personal computers will mainly be subscribing to the information on the gateway and not performing any intense processing with it.

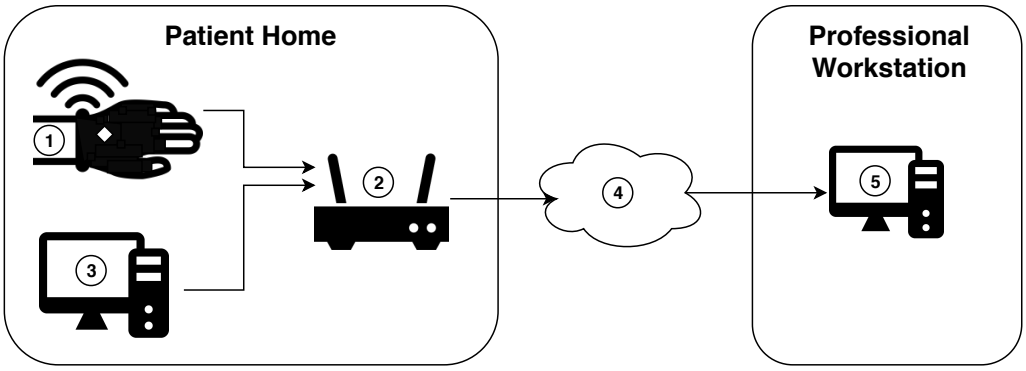


Figure 2.1: Architecture of UPDA System

2.2 Customer Needs

We conducted a wide variety of research in order to gauge the potential market and determine customer needs. We read publications to familiarize ourselves with the current state of the market, and then we began to survey Parkinson's Disease physicians and patients to answer any further questions and concerns [11, 6, 2]. We used equation 2.1 to calculate our opportunity to satisfy different customer needs and it is presented in Table 2.1.

$$Opportunity = 2 * Importance - Satisfaction \tag{2.1}$$

Table 2.1: Customer Needs and Opportunity

Customer Desired Outcomes	Importance	Satisfaction	Opportunity
Monitor PD Periodically	10	3	17
Automatic UPDRS Scoring	10	0	20
Better Diagnosis Between ET and PD	8	2	14
More Accessible UPDRS Testing	5	3	7
Open Source Tremor Data for Research	7	5	9
Long-Term Tremor Monitoring	8	2	14
Freeze of Gait Detection	10	3	17
Low-Cost Solutions for Tremor Management	10	6	14

2.3 System Level Requirements

From our research and exploration of customer needs, we determined our system level requirements for the basis of this project.

1. Automated UPDRS Evaluation

    1.1. Assess Parkinson's Disease tremors based on several tests from the UPDRS Motor examination

without the immediate presence of a physician. The tests performed are as follows:

- UPDRS 3.4 Finger Taps
- UPDRS 3.5 Hand Movements
- UPDRS 3.15 Postural Tremor of Hands
- UPDRS 3.16 Kinetic Tremor of Hands
- UPDRS 3.17 Rest Tremor Amplitude
- UPDRS 3.18 Constancy of Rest Tremor

1.2. The system will assess tremors noted in requirement **1.1** by analyzing the patient's regular daily actions in the home or identifying choreographed test on its own.

2. Usability

2.1. The wearable portion of the system will be worn on the patient and collect data without inhibiting the patient's regular movements during their daily routines.

2.2. The system will require the minimum effort on the part of both the patient and doctor for setup, running, and maintenance.

3. Environment

3.1. The system will run in either a hospital environment or a home environment

3.2. The system will focus on energy efficiency during through both the design of the software and hardware.

4. Reporting

4.1. The system must evaluate raw data and produce UPDRS scores by the end of the recommended period which the system is utilized by the patients.

4.2. The system will store and display raw data, the extracted features from the raw data, and unsupervised UPDRS scores.

4.3. The patient's data will be treated with the appropriate levels of confidentiality during inter-system communication as well as from the patient's system to a doctor's/researcher's system.

2.4 Bench-marking Results

There has been a resurgence of interest for better monitoring the health of patients [6, 12]. These professionals have reviewed all devices in publications regarding the measurement and evaluation of different

Parkinson's Disease symptoms. When reviewing and evaluating each device, the researchers look at several different features of the use of these devices to categorize them into several categories from best to worst: recommended, suggested, and listed. Of the devices in the recommended category, two devices were cited that are capable of measuring tremors [6].

2.4.1 Kinesia

Kinesia is a hybrid device, meaning that it requires both a wearable device as well as a remote computer to perform its measurement and evaluation procedures on a patient. To measure a patient's Parkinsonian tremors, the Kinesia uses accelerometers and gyroscopes built into a ring, which communicates with a computer on a wristband. Figure 2.2 shows an image of one of the older models with the finger ring. This ring, accompanied with a bracelet, can be used to measure and quantify tremors. This device then transmits the data to a local computer. This device is geared towards quantifying tremors, assessing dyskinesia, and quantifying bradykinesia [6].

Great Lakes NeuroTechnologies offers several different versions of the Kinesia device for both home monitoring as well as for clinical trials. The device has reported oustanding performance, but has received poor user acceptance, since 45 percent of users have noted that they would not wear the devices in public [6]. Additionally the system in place for UPDRS scoring requires a tablet to be set up and controlled, which may be difficult for someone with advanced PD.

Figure 2.2: Kinesia Hybrid Wearable Device

2.4.2 Physilog

The Physilog is a wearable device that focuses on measuring gait, sway, tremor, and bradykinesia [6]. It utilizes an inertial measurement unit embedded within a small container, that can be strapped to several different locations on the patient depending on the type of measurement that is required. Figure 2.3 shows an image of the most recent Physilog sensor attached to a user's foot to model gait.

The Physilog device is commonly used to detect gait features in patients, but has been used to determine tremor. Furthermore, this device has been used in a study to measure and quantify tremor and bradykinesia with successful results. While the results are impressive, the daily actions that are measured still seem constrained. The Physilog device additionally provides application programming interfaces (APIs), which allow the user a finer level of control of the device, making it suitable for research.

Figure 2.3: Physilog Wearable Device

2.5 Functional Analysis

This system has several subsystems that together are responsible for the collection of raw data and generation of UPDRS scores. Figure 2.4 shows a visual of the pipeline and the ordering of the subsystems. The wearable subsystem will be responsible for the collection of raw data from the patient. The server subsystem is responsible for collecting the raw data from the wearable and preparing the data for filtration. The filtration subsystem then uses the raw data to synthesize new features, which are used for the position determination and scoring subsystems. The position determination examines the data for key sampling periods, which could determine a score. The scoring subsystem then takes the information generated from the other subsystems and produces a score and a score report.

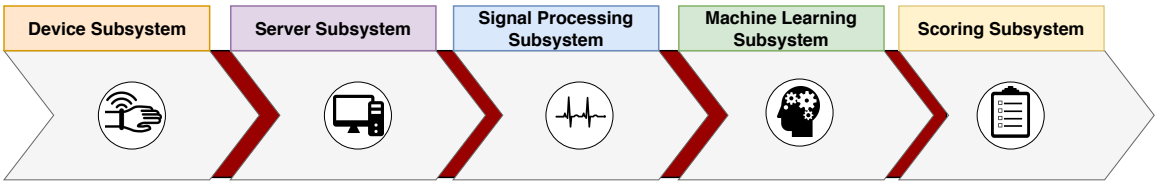


Figure 2.4: UPDA System Pipeline

2.5.1 Functional Decomposition

This subsection describes all of the functions as well as the required sub functions for each. These functions will be described more in detail within each of the subsystem chapters to which they belong.

1. Data Collection
    1.1. I2C communication with MPU 9250
    1.2. Analog sampling from Myoware sEMG
    1.3. Interrupt driven sampling
    1.4. Instance data buffering
    1.5. Instance data storage
2. Data Transfer
    2.1. Extract data from storage
    2.2. 802.15.4 Packet preparation and fragmentation of data
    2.3. Packet transmission and attempted QoS over radio
    2.4. Packet reception, de-fragmentation, and server storage
    2.5. Patient profile preparation
3. Signal Analysis
    3.1. Pipeline triggering and initialization
    3.2. Data retrieving and matrix formation
    3.3. Synthesized Data Storage
    3.4. Mahony Filtration
    3.5. Low-pass Filter
    3.6. Bandpass Filter

2.5.2 Device Subsystem

The device subsystem is responsible for the collection and transfer of the patient's raw data. This subsystem uses embedded electronics attached to a glove to monitor a patient's data. The glove transfers raw data to the server where it is used to produce UPDRS scores. The device measures patient activity using four inertial measurement units (IMUs) and one surface electromyography (sEMG). These sensors will all be sampled and sent for processing via a microcontroller (MCU) that will be stationed on the console of the device.

2.5.3 Server Subsystem

The server subsystem is responsible for the integration between the wearable device and the subsystems that score a patient's data. The server is able to currently form a link with a device and store patient data transferred from it. At the user's request, the data is then sent to the pipeline for processing.

2.5.4 Signal Analysis Subsystem

The signal analysis subsystem is responsible for processing raw data and preparation of the signals for the machine learning and scoring subsystems. The raw signals are filtered to reduce noise or focus on specific frequencies. This subsystem is also used to synthesize new features such as orientation, which are essential for scoring.

2.5.5 Machine Learning Subsystem

The Machine Learning Subsystem is responsible for recognizing and quantifying certain significant actions that occur in a patient's daily movements that help to establish a UPDRS score. Particularly, it focuses on the first two tests, Finger Taps and Hand Movement. Our system uses a machine learning algorithm to estimate whether or not a patient performs a certain qualifying action without the need for choreographed movements. The role of the machine learning subsystem is to classify a patient's normal movements into different categories representative of separate Machine Learning Models. We train the models using a variety of different Machine Learning methods and test the results to see which model gives us the best results.

2.5.6 Scoring Subsystem

The scoring subsystem is responsible for gathering the data from the previous subsystems and utilizing them to generate a UPDRS score. Additionally, the scoring subsystem carries the reporting functionality, which pulls the results from the scoring subsystem and the original data to generate reports.

2.6 Team and Project Management

This section describes the organization of the team and project management strategies implemented to make the project go smoothly.

2.6.1 Challenges and Constraints

This project was extremely ambitious, and as a result time management of our project was arguably one of the most challenging aspects. Given the short development period, we had to spend our time in such a way where we could quickly test our project.

Additionally, several of the subsystem relied on the device subsystem performing during the early stages of development, which was extremely stressful. The short development time span required conflicted greatly with the research based nature of the project, limiting our abilities to conduct more thorough tests or gather enough data to improve our algorithms.

One of the biggest management tasks we had were to ensure that the deliverable for both the Bioengineering Department and Computer Science and Engineering Department were satisfied.

2.6.2 Budget

Our main consumption of the budget is based mainly of hardware for development as well as materials for human testing. The detailed budget plan is listed in table 1.

2.6.3 Timeline

In figure 3, is a description of the project's timeline over the fall of 2017, winter of 2018, and spring of 2018. Each quarter has a set of deliverables that must be delivered throughout the quarter. Alongside each deliverable are different technical tasks that must also be completed.

There are three general stages that make up the development of the project. The first stage will involve the prototyping and testing of the wearable device, which will take place during the Fall as well as over Winter break. Alongside wearable prototyping, we will start algorithm prototyping, which will produce pseudo code for the all the important algorithms in the system during winter break. The second stage during the winter quarter will include the connection of the wearable device to the server for analysis and begin the process of integration. While this is happening the wearable prototype will be used to start collecting data to test and improve the performance of the server-side algorithms. The third stage, during the spring quarter, all of the different features of the system will be put together and tested from end to end. If there is time available, further tests and optimizations will continue on the project to improve accuracy and efficiency.

2.6.4 Risks and Mitigations

We created a risk analysis table in order to evaluate each risk that could occur during the development process. This table includes the consequence, probability, severity, impact, and method to mitigate each individual risk.

Implementation Risks

Implementation risks originate during the implementation period that can affect or compromise the project.

Risk	Consequences	Probability	Severity	Impact	Mitigation
Group member gets sick	Delays in completing the system	0.5	1	0.5	Vitamin C, sleep, and situational awareness
Time	System not entirely finished by deadline	0.1	3	0.3	Schedule accordingly
Bugs	System does not work properly	0.07	3	0.21	Include comments to improve debugging process
Lack of understanding the system requirements	Delays in completing the system	0.01	3	0.03	Clarify system expectations with client
Data Loss	Losing our work	0.002	4	0.008	Keep backups of our data on Github

Figure 2.5: Risk Analysis Table

Technological Risks

Technological risks originate as a result of the utilization of the different technologies in the system.

Risk	Consequences	Probability	Severity	Impact	Mitigation
Battery Failure	Harms Patient or System	0.01	10	0.1	Prioritize patient safety over performance
Inaccurate Sensor Readings	System cannot perform adequate diagnostics	0.4	5	2	Create robustness in sensor data through sampling, multiple, and placing importance on high-quality sensors
Wearable Hardware Failure	System cannot read data	0.2	7	1.4	Have the system die gracefully and have extra hardware on hand for backup during development
Networking Hardware Failure	System cannot communicate data	0.2	7	1.4	Have the system die gracefully and have extra hardware on hand for backup during development
Poor Network Connectivity	System cannot communicate data to doctors or researchers	0.4	4	1.6	Design the system in a way that both endpoints with opportunistically connect with each other.

Figure 2.6: Technological Risk Analysis Table

User Risks

User risks involve different potential errors that can arise if the patient misuses the system or is harmed by the system. This also includes users such as the doctors and researchers that may be reading information from this system.

Risk	Consequences	Probability	Severity	Impact	Mitigation
Patients forget housekeeping of the wearable system	System will not have enough power to collect patient data for analysis	0.5	10	5	Create a reminder to promote charging after use
Patients are unable to understand how to use the system	Patient's PD data cannot be analyzed and assessed	0.5	4	2	Design a simple "push and play" system
Patients break the wearable system	System is unable to collect patient's PD data and analyze it	0.5	10	5	Quickly alert users that the system is broken
Doctors/Researchers do not understand how to use the system	Doctors will not be able to confirm automated patient UPDRS scores and adjust medication effectively	0.5	4	2	Have a tutorial for the doctors and researchers to follow
Patient system looses communication with the Doctors/Researchers systems	UPDRS analysis cannot be sent to doctors or researchers	0.5	2	1	Have the system store/throw away data and continue attempting to connect to the system

Figure 2.7: User Risk Analysis Table

2.6.5 Team management

This sections describes the different strategies employed to manage our team. This section includes discuession of the general team dynamic and our method for coordinating development.

Team Dynamic

The team dynamic is best described as democratic and centralized, which most suits the work ethic and communication. Having a democratic and centralized team means that each person can contribute their ideas to the group freely, but there is one member driving the conversation usually to make sure that the group does not go off track.

Development Method

Due to the device-driven nature of the project, we utilize a combination of Scrum and Kanban development methodologies to finish work. More specifically, the team performs weekly sprints, where each sprint's tasks and progress are recorded on a Kanban board. Appendix figure 4, shows a week's sprint organized on a Kanban board on Github. For each weekly sprint, the team meets three times outside of personal work. The first day of the week it to take care of all the "housekeeping". The first day the team checks the progress from

last week's sprint and creates a new sprint for the upcoming week. The second day is a light work day, where the team meets up to discuss important design decisions or work out small bits of the project. All major work for the project happens on the third day, when the team meets for a five-hour work session for coding and device development.

Chapter 3

Wearable Device Subsystem

The wearable device subsystem is responsible for the collection and transfer of a patient's raw data for UPDRS scoring. As seen in figure 3.1, the device has a glove element and a console element. The glove element houses the IMUs, which track orientation. The console portion houses the sEMG for monitoring muscular activity, the radio, the MCU, and the rest of the main circuitry required for the system to work.

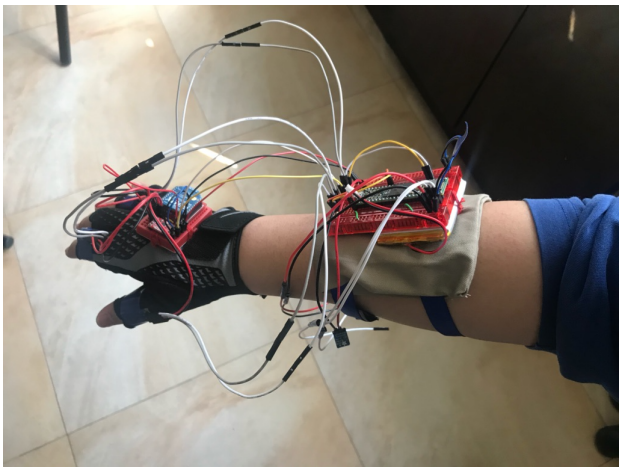


Figure 3.1: Wearable Device Prototype v3

3.1 Subsystem Requirements

- Measure inertial and orientation data of the patient's fingers with a precision of at least two-hundredths, to evaluate all required finger tests.
- Measure raw sEMG information of the patient's forearm to provide another layer of information to measure and evaluate all specified MDS-UPDRS tests.
- Successfully transfer information from the wearable device to the server end securely with minimal loss in packets, so that data can be processed on the server end.

- Wearable device and server should follow standards associated with first class medical devices.
- Sensor placement should be non-intrusive for a patient performing regular daily activities.
- Wearable system and server should require minimal effort for setup, initialization, use, and maintenance on the part of both the patient and the clinician.
- Wearable system must be conservative with battery consumption while in use.

3.2 Trade-offs

This section examines the different design decisions made for the device subsystem and their implications on system performance.

3.2.1 Method of Patient Monitoring

There several methods that have been explored by other teams to monitor and collect a patient's health data [6]. These methods are divided depending on the distance and level of interaction with the user. The main examples of long-distance forms of measurement such as cameras. An example of closer proximity sensors would include any type of wearable sensor or device.

The system requires the capability to collect a continuous stream of information from Parkinsonian tremors through the patient's day. Using a long distance measuring device does not inhibit the user, but it is impractical for continuous data collection when the user is not constrained by area. Furthermore, the use of long distance data has a tendency to be less accurate than desired for medical diagnostics. A comfortable wearable is constantly on the patient, meaning it can continuously stream data for analysis and due to its close-proximity has a higher likelihood for more accurate measurements.

3.2.2 Method of Wireless Communication

A major feature of the system is the form of communication used to send the data between the device and the server. Wireless communication is used to ensure no restrictions on the patient's movement. The three most popular technologies for wireless transfer are 802.11 (WiFi), 802.15.4, and 802.15.1 (Bluetooth).

The common use case for the system would likely be a patient wearing the device in home, which communicates with a local router to talk to a server. We determined that the system should choose a technology that supports medium range distances reliably with a sufficient data rate.

802.15.1, while capable of transferring data sufficiently, is only effective at low powers at a close proximity. Consequently, 802.15.1 was removed from further consideration. Figure 3.1 compares 802.15.4 and 802.11 in terms of range, the data transfer rate, and the power consumption. While 802.11 can support more

data this feature would come at the cost of extremely high power consumption, which is not ideal for a wireless medical device. Each sample is close to the carrying capacity of an 802.15.4 packet, and the system is willing to accept the the possibility of some packet fragmentation at the benefit of much more efficient power consumption.

Table 3.1: 802.15.4 versus 802.11

	Range	Data Rate	Relative Power Consumption
802.15.4	20m	54 mb/sec	high
802.11	100m	250 kb/sec	low

3.2.3 External Storage

The device must be capable of reliable communication with another computer over a local network, but in the case of unreliable connections the device should be able to save data. Instead of throwing away useful data in this event, the micro controller should be able to save the data on a permanent storage device and send it to the other computer once the connection is re-established. Since the system relies on having detailed measurements on the patient's actions, every opportunity for data salvation should be taken.

3.2.4 Microcontroller

The microcontroller (MCU) coordinates all the other hardware elements on the wearable. Specifically, the device uses an ARM Cortex-M4 MCU (Kinetis, MK66FX1M0VMD18), which uses a RISC-based architecture with a normal clock rate at around 180MHz. Using interrupt-based timers, the MCU is capable of sampling data in real-time reliably while maintaining a low power profile. Any major computations are passed off to the server, allowing the device to focus mainly on sampling and power consumption.

For prototyping simplicity, the ARM MCU is built into the Teensy 3.6, which comes with several additional advantages. This device is compatible with Arduino programming files, meaning that we can develop the software for the device in a similar environment. The Teensy also has attached a built-in micro SD card interface, providing easy external storage, which is essential for the device prototype for data collection and logging.

3.3 Design

This section will describe the design of the device as well as analyze these designs to understand their implications. First we will discuss the general design of the system in terms of circuitry, physical design, and code. Then we explore how much data we send, and the costs of sending data over the 802.15.4 radio.

Finally we will explore the limitations of the device in terms of energy usage, and how this can be extended using code.

3.3.1 Physical Design

We are focused on patients from all stages of Parkinson's Disease, so we picked a glove design for the device to reduce the interference between the user's actions and the device.

The UPDA system focuses on data collection to evaluate finger and arm based UPDRS tests. These tests require the detection of several actions including the movement of individual fingers and position of the entire hand. Specifically, these tests require the detection of several actions: pointer finger and thumb pinch, hand close, hand reaching straight out.

We mounted our sensors onto a glove in order to best cover important actions without interfering with the patient's intentional movement. On the glove four inertial measurement units (IMUs) were placed on the proximal phalanges of the thumb, the index finger, ring finger, and the dorsum of the palm (Figure 3.2). Separate from the glove is a additional sensor for sEMG, placed on the patient's forearm to measure muscular activity.

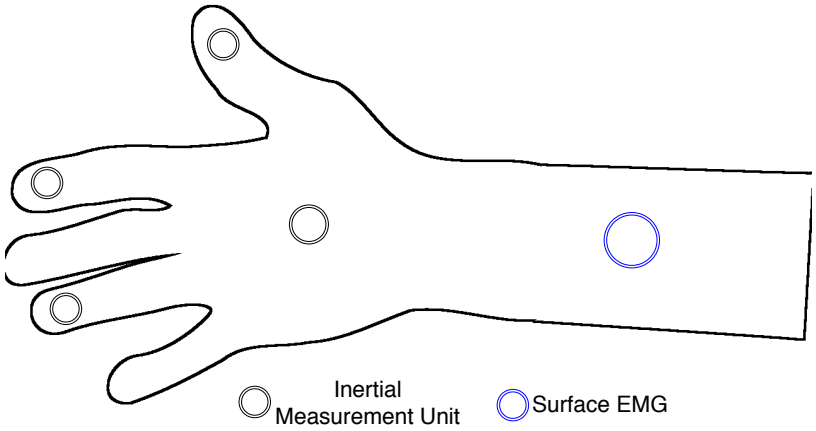


Figure 3.2: Conceptual Model for Wearable Device Sensor Placement. Inertial Measurement Units are placed on the dorsum of the palm, the thumb, the pointer finger, and the ring finger. The surface EMG is placed on the muscle belly of the forearm.

3.3.2 Circuit

In the appendices, figure 1 shows the current circuit design of the device. The battery is connected to the Teensy microcontroller, which powers and communicates with the rest of the hardware.

Figure 3.3 displays the relationships and connections between the different hardware. The Teensy collects information for the sEMG over an analog input, which passes the analog information through an 13-bit analog to digital converter, making it usable by the MCU. The IMU sensors has to communicate with the MCU over

two I2C buses to ensure that each sensor has its own unique address. Each sensor can only assume two different addresses, meaning that two I2C busses are necessary in order to speak with each sensor without confusion. To ensure reliable signals, a pull-up resistor is added to each clock and signal.

The Teensy communicates with the 802.15.4 radio over an Universal Asynchronous Transmitter and Reciever (UART), which requires only a transmission bus and a receiving bus. However, in addition to wiring the radio, it must be first configured before it can be used correctly. Both the MCU and the radio must agree on a baud rate with which to communicate. In the event that the two hardwares do not share the same baud rate, transmitted data will likely look like gibberish from the receiving end.

While this device is functional when plugged into a 5 volt power supply, the Teensy has repeatedly had problems reading and writing to its micro SD interface while on 3.7 volts. After some research, we have learned from other developers that the circuitry regarding detection of the micro SD card is fragile and unstable at lower voltages. In the future, we full expect to adjust this circuitry to avoid this problem.

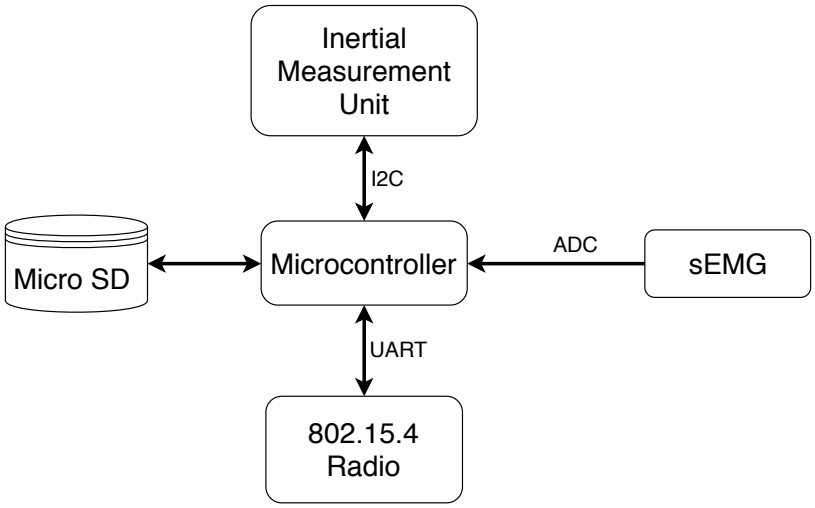


Figure 3.3: Hardware Architecture for UPDA Glove

3.3.3 Code

For the phase one prototype, the device behavior is defined best as a simple producer-consumer system. Figure 3.4 shows how the device supports a sampling mode for data collection and a transfer mode to data transmission. The user can select which mode to be in by pressing the button on the wearable device, which will cause an interrupt in the MCU which then switches modes.

Sampling Mode

After the device has been turned on and all of the hardware has been initialized successfully, the MCU creates a circular buffer for samples. The circular buffer, is implemented specifically as a queue. The cir-

cular nature of this specific queue means that the buffer is more efficient is how it stores and removes its information.

Once the circular buffer has been created, microcontroller creates a timer interrupt for the sensor interrupt service routine, which forces the microcontroller to read information from its sensors every one-hundredth of a second and store the information into the buffer. The main thread of the MCU will continuously remove items from the circular buffer and store them on the micro SD card to ensure no data is lost by performing real-time data transfer.

Transfer Mode

When the patient has finished collecting data, the patient presses the button again to switch the device back into transfer mode. In transfer mode, the device searches through its permanent storage to see if any data has been collected. If data is found and the device is online, the device will initialize the data transmission session with the server. The explicit protocols defined for this interaction are defined in the design section of the server subsystem.

Before sending data, the device has to prepare it for transmission. In order to do this, the device parses the raw text and extracts the data needed to create a new packet to send to the server. The device will then attempt to send the packet to the server and wait for an acknowledgement. If no acknowledgement is received, the device will wait half a second before continuing to re-send the packet for an acknowledgement. After one-hundred tries, the radio will assume that the link between the server and the device is down, and the device will save the transmission of packets for the next time it has linked with the server. While the device is in transfer mode but cannot find the server, the device will sleep and wake up occasionally to broadcast, searching for a local server.

Power Consumption Behavior

The device has employed energy saving strategies to extend the lifetime by turning off different elements of the device when they are not in use. For instance, the device will turn off the radio while in sampling mode and while in transfer mode, the device will turn off the IMUs. However, if there is no data to send, the device will turn off the MCU and the IMUs until the user decides to collect more data. Equation 3.4 estimates the battery life of the device where each hardware element is idle, which provides a good upper limit to the performance of the device. While testing the lifetime based on normal usage of the device, we can estimate that the actual lifetime of the device will fall somewhere between the two extremes calculations.

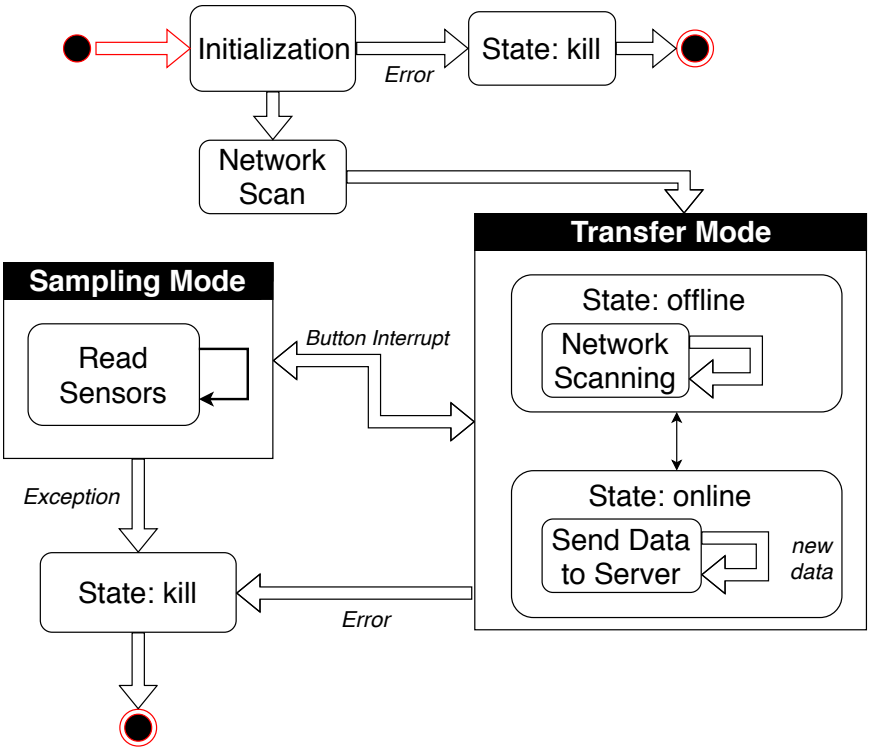


Figure 3.4: Behavior of UPDA Device

3.3.4 Analyses

For the wearable part of the subsystem, there were different analyses done to measure the longevity of the device as well as its capabilities to transfer the information it has collected.

Bits Transferred Per Packet

An important element to this system is the 802.15.4 radio, which can only transfer 250 kilobits per second. In addition, each sample packet transferred requires additional frame data to encapsulate the information and coordinate transfer, reducing the available bits down even further. Given that the system must sample at least 100 Hz, calculations must be done to ensure that the information transferred will fit these restrictions.

In total, a device has four inertial measurement units (IMU) and one surface electromyographic sensor(sEMG). Each IMU produces nine features that are deemed necessary for transfer and each sEMG produces two features. Given that the sEMG features are integers and the IMU features are floats, figure 3.2 can be used to calculate the size of a payload per packet. Given that we are producing 118,400 bits per second, our 802.15.4 radio should be able to handle the transfer of the data over the radios.

$$Payload(bits) = (2 \times 16) + 3 \times 4 \times (3 \times 32) = 1,184(\frac{bits}{sample}) \tag{3.1}$$

$$Payload(\frac{bits}{sec}) = 1,184(\frac{bits}{sample}) \times 100(\frac{sample}{sec}) = 118,400(\frac{bits}{sec}) \tag{3.2}$$

Table 3.2: Data Type Sizes on Teensy

Data Type	Size (Bits)	Size (Bytes)
float	32	4
int	16	2
char	8	1

Device Lifetime

Since the device is interested in the measurement of symptoms over a period of time, it is important to analyze and estimate the lifetime of the device to increase our ability to collect data points. Automatic UPDRS scoring requires for the continuous sampling of Parkinsonian symptoms over a long time to increase the probability that testable actions will occur.

Table 3.3: Approximate Power Consumption of Major Wearable Hardware

Device	**Active Current Draw**	**Idle Current Draw**
Teensy 3.6	79.13 mA	60 mA
XBee S1	50.0 mA	10 µA
MPU 9250	3.7 mA	8 µA
sEMG	14 mA	N/A

$$\frac{2500mAh}{79.13\text{mA} + 50\text{mA} + 14.8\text{mA} + 14\text{mA}} = 15.738\text{h} \tag{3.3}$$

$$\frac{2500mAh}{60\text{mA} + 0.010\text{mA} + 0.032\text{mA} + 14\text{mA}} = 33.76\text{h} \tag{3.4}$$

Given a lithium ion battery rated at 3.7 volts and 2500 milliamp hours and figure **??** showing the operational current of each device, we can calculate for the total lifetime of the device assuming that all aspects of hardware are running constantly.

According to calculation 3.3, the system should be able to last fifteen hours running constantly, which matches the systems needs in terms of longevity to collect enough data. To further extend the lifetime, the system can be optimized to save power and use costly hardware like the radio sparingly.

Assuming that the device is constantly asleep, we calculate a best case performance of approximately thirty three hours for the system as seen in equation 3.4. The actual lifetime of the device will contain a mixture of active power consumption and idle power consumption. Therefore the expected lifetime should

be somewhere between approximately fifteen hours and thirty three hours. Since we expect only to measure information for a couple of hours a day, this battery life will definitely be suffiencient for use.

Chapter 4

Server Subsystem

The server subsystem manages the interaction between the device subsystem, the signal processing subsystem, the machine learning subsystem, and the scoring subsystem. The primary goal of the server subsystem is to receive the device's data and transform it into a suitable format for the pipeline to initiate automatic UPDRS scoring.

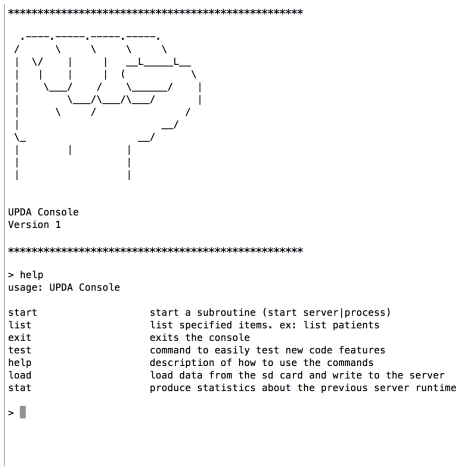


Figure 4.1: Screenshot of UPDA Server Console. Customized console displaying all the user-defined commands for server control

4.1 Subsystem Requirements

- Hosts a 802.15.4 server which can receive information sent from a specific wearable device.
  - The server can respond acknowledge device messages
  - Capable of combining fragmented messages sent from the device.
  - The server has the ability to differentiate between different sets of data produced by devices.
- The server can be customized for research use and further development.

- The server can be tested in the event of wireless failure
  - The server can generate patient profiles by loading the micro SD card into the computer
- The server can perform automated UPDRS scoring on a patient's profile

4.2 Trade-offs

This section discusses the pros and cons involving different data transfer strategies and server choice.

Data Transfer

A big factor for how the information will be processed is determined by how the information will be transferred from the wearable device to other computers. Usually a device can either communicate over a wired connection or a wireless connection. The use of a physical connection in this system would be feasible, but limits a patient's freedom, which is important for a patient to perform daily actions.

The wireless connection is more desirable for several reasons. A wireless communication has a much longer range without inhibiting the user. The data collected can be sent at periodic times during the patients use instead of at the very end, reducing the amount of work the patient has to do.

As a result, the devices chosen for this system must support wireless communication. Since the 802.15.4 communication standard is being used for this project, a 802.15.4 compatible server will be used to communicate with the wearable device.

Server Choice

A big trade-off in the design consequently came from the choice of using an 802.15.4 radio connected to a laptop as a server versus an Artik gateway. We have decided that while using the Artik gateway would serve as a more realistic application of the device, the laptop would allow for better development flexibility for debugging and testing.

4.3 Design

This section will describe the design of the server. For the purposes of this proof of concept, the design of the server has been kept fairly simple. The server hardware consists of a radio and a laptop interfacing together using the USB protocol. The laptop is running a server program, which triggers callback functions when the radio receives incoming messages. When a message is received the server unpacks the packets and passes them into the pipeline for UPDRS scoring.

4.3.1 Analyses

This section provides supporting calculations of the specifications of the system, details of its analysis, as well as the results of the system.

4.3.2 Code

The server code is responsible for receiving any incoming transmissions from the device and pass them onto the pipeline for processing as seen in figure 4.3. The server is written in Python so that we could utilize the many quality data processing APIs designed for Python.

Console Graphical User Interface

To improve the ease of use of the server, we decided to add a command line interface, that will allow a user to initialize different parts of the server for testing and gather statistics on the data. Figure 4.1 shows the command line interface and table 4.1 explains all the current commands available currently.

Table 4.1: UPDA Server Console Commands

Command	Description
start	start a subroutine (start server/process)
exit	exits the console
test	command to easily test new code features
list	list specified items. ex: list patients
load	load data from the sd card and write to the server
stat	produce statistics about the previous server run time
help	description of how to use the commands

Included in the server software is a Console class, which enables a programmer who is knowledgeable in Python to design a basic console with any command necessary. In the future, researchers can use these simple classes to collect data and perform functions required by their processes, instead of feeling constrained by any server API set in place.

Communication Protocols

In order for the server and the device to communicate effectively about data transfer, we used a simple protocol to establish connections, signal incoming new data, and the end of a patient's data session. Figure 4.2 shows how information is placed into the 100 byte 802.15.4 packet.

After the header, the first byte of the packet sends is the method, which defines what type of information is being sent. The methods are: broadcast, new data, continue data, and payload.

The device sends the ”broadcast” method when the device wants to connect with a local server and then waits for an acknowledgement to notify the device that it is connected. A ”new data” method or ”continue

data" method is sent if the device is attempting to transfer a new set of data sampled by the patient or continue sending an unfinished set of data. Finally, the device sends the "payload" method whenever sending patient data.

The second byte, "Device ID", specifies the identification number of the device. This allows the server to differentiate between messages from different devices. Since the server only supports one device at a time at the moment, other device messages will be dropped if it is already linked with a device.

The third field, "Payload ID", identifies the packet sent. This is important because in order for a sample to be transferred to the server, it must be fragmented due to the 100 byte constraint on the radio's payload. The server then uses the payload ID to combine fragmented packets back together on the server side. In the payload field, all of the sampling data is encoded into the payload using bits to preserve as much space as possible.

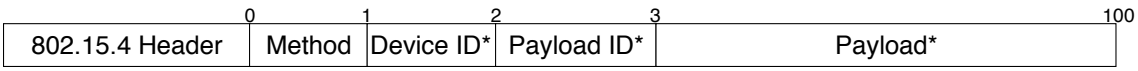


Figure 4.2: 802.15.4 Payload Packet Organization

Server Processing

The first portion of the server is responsible for listening to the 802.15.4 radio for incoming packets. To do so, a thread is created that utilizes a callback function, triggering each time a new packet comes in.

When a first broadcast packet is sent by the device, the server acknowledges the broadcast to tell the device that it will process its data. The server then will create a new data structure to monitor the link between the device in question and the server. When the device sends a packet with the "new data" method, the server will create a new patient profile for the device's data to be stored into. Patient profiles are stored as folders with the patient's identification number, which store all of the files pertaining to the data of that patient. At the new arrival of a payload packet, the server thread de-fragments the packets and extracts the floating point numbers from the binary encoding. While extracting the data, the server will also estimate the orientation of the patient's hand using the Mahony filter, given the position data at hand. All the extracted data is then stored on disk in a text file in the patient's profile.

Pipeline

After the data from the device has been collected, the server will then pass it through the pipeline to perform automatic UPDRS scoring.

The server code serves as the entryway into the entire data processing pipeline. In the appendices, figure 4.3 shows the pipeline process. The server waits for information from the wearable device. Once it receives

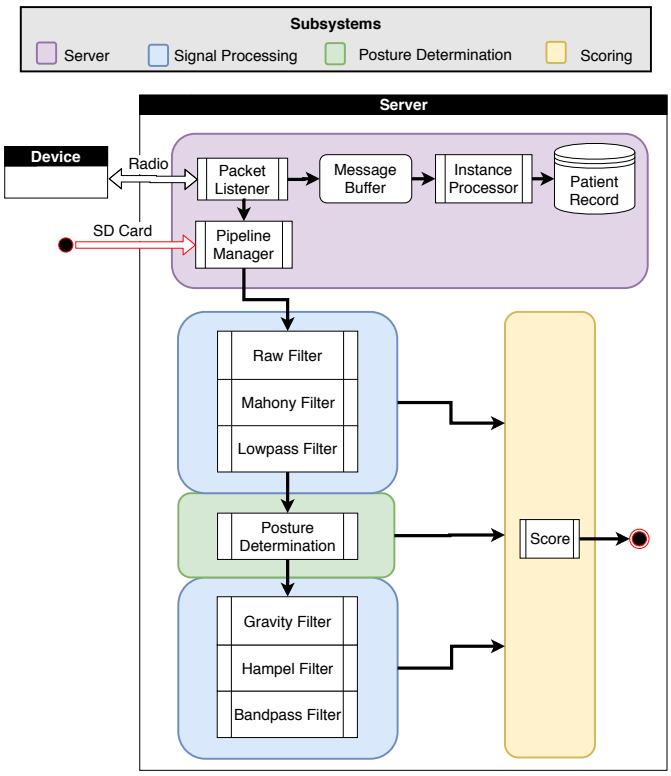


Figure 4.3: Data Processing Pipeline on Server

the sampling data it will be passed through several functions to filter the signal to evaluate PD tremors.

To pass the data through each filter, a Pipeline Manager object is created. The Pipeline Manager class defines an extra thread, which is assigned a patient profile to score. Since Pipeline Managers are threads, it is possible of the infrastructure to score multiple patients at a time, which will improve the platform's scalability. The Pipeline Manager thread first takes the patient's raw data and pass several specific channels of interest through the low pass filter. Next, the manager passes all raw data and generated data through the posture determination module, which utilizes the machine learning subsystem to determine important positions for UPDRS scoring. After this, the Pipeline Manager passes the rest of the data through gravity filter, the hampel filter, and the bandpass filter to produce the final results. Finally, the Pipeline Manager calls the scoring algorithm, which pulls data produced from all the different filters and produces a score that is recorded by the reporting module.

File Per Filter Philosophy

While processing filters, we made the explicit choice to produce a new file the patient's profile for each filter used. Alternatively we can reduce the amount of hard disk space we use by storing all of the new data on RAM, where it can more easily be recalled when needed. For example, all of the filtered data produced

from the lowpass filter would be stored in a file called ”lowpass.txt” under the patient’s profile on the server. While storing all results on RAM may seem like a more efficient method initially, we believe that creating new files for newly synthesized features is a better long-term option.

As the system grows, it will be required to process more and more data. Therefore it is important to adopt the methodology of pulling information in from disk as needed instead of trying to extract all the data at once. A patient’s synthesized data may eventually not even fit in RAM. Producing an output for each filter also has proven to be an excellent debugging strategy. The outputs allow us to check the quality of each filter more easily and determine if the algorithms are functioning to our standards or not. If data was instead only held in RAM, we would likely need to write it to disk anyways to load and graph the data in MatLab.

Chapter 5

Signal Manipulation and Analysis

5.1 Introduction

The goal for signal manipulation and analysis subsystem is to process all raw data collected from the device to produce meaningful features, which could be further analyzed for diagnostics propose. In the scope of this research, we aim on six tests from UPDRS, which cover fingers and hands movement. Therefore, we need to process acceleration, angular velocity and sEMG readings on fingers and hands to generate features that can help determine UPDRS scope of these six tests.

The first two tests are analyzing finger taps and hand movement, which requires the system to generate features that can reflect patient's desired movement without tremor. We first implemented a low-pass filter to clear high frequency noise caused by patient's tremor and device itself. In addition, we created a gravity filter to correct raw acceleration, where gravity acceleration is added to the signal.

The other four tests are about different tremors and their constancy. We use positional data from previous analysis and sEMG signals to differentiate the time of posture tremor, kinetic tremor, and resting tremor. We designed a band-pass filter to isolate tremor signal to analyze amplitude and frequency. To obtain clear sEMG signals, we use hampel filter to clean the outliers caused by poor contactability between sensors and skin.

5.2 Options

In this section, we will discuss key options and our choices we made for analyzing signals.

The first option we have is whether we should make the filters by electronic components or by algorithms. The advantage with electronic filters is that they can perform real-time analysis and the performance is relatively stable. However, they are not easy to alter and are not adjustable for customized design. On the other hand, digital filters have the advantage of easy to edit and design for specific usage. In addition, there are more options of digital filters for us to choose in order to perform different analysis. Therefore, we decided to use digital filters in stead of electronic filters in our entire design.

The second option for us is which digital filters we need to include in our design to fulfill the requirements. We explored over ten different digital filters and finally decided to use low-pass filter, bandpass filter and hampel filter to analyze our raw signals. Furthermore, we created a new filter algorithm called gravity filter to adjust the gravity effect our raw acceleration.

5.3 Method

In this section, we will discuss the implementation of different filters, which produce key features to perform UPDRS scoring. We designed four different filters to apply to different raw signals collected from our sensors. We use a low-pass filter to prepare data for machine learning, a band-pass filter to identify tremor, a gravity filter to correct gravity effect on acceleration, and a hampel filter to eliminate spikes on sEMG signal.

5.3.1 Tremor Model

We currently have limited access to Parkinson's Disease patients, so we create tremor model by intentionally shaking hands by healthy patients. We must first validate the tremor data before using it for further analysis and training our pattern recognition model. The frequency of a typical Parkinsonian rest tremor is above 4 HZ and can reach 9 HZ [13]. We applied a Fast Fourier Transform (FFT) on measured data to analyze our model tremor and determine whether our modelling is sufficiently accurate for further analysis.

5.3.2 Filter Design

We designed two digital filters to extract the desired signal from the raw data collected from glove. A low-pass filter was designed to isolate the patient's desired actions, meaning actions without tremors. Since a typical Parkinsonian tremor is above 4 HZ [13], we set the the cut-off frequency to be 3 HZ and stopband frequency to be 3-4 HZ. We choose the FIR instead of the IIR response filter for better performance on finite samples because we perform data analysis after data collection. A band-pass filter was designed to isolate tremor signal from raw data. We chose 3-7 HZ as our passband to include all tremor information from model tremor we generated. The parameters of our preliminary filter designs are listed in table 5.1.

We use Filter Builder in MATLAB to build these two filters based on the parameters listed in table 5.1. We then output the filter coefficients as a text file and import them it into server. We perform convolution between the raw signal and the filter coefficient array to obtain filtered signal using the following equation:

$$(Signal * Filter)(t) = \int_0^\infty Signal(\tau) \cdot Filter(t - \tau) d\tau \tag{5.1}$$

Table 5.1: Filter Parameters

Filter	Low-pass Filter	Band-pass Filter
Filter Type	FIR	FIR
Order Mode	Minimum	Minimum
Passband Frequency	0-3 HZ	3-7 HZ
Stopband Frequency	3-4 HZ	2-3 HZ; 7-8 HZ
Passband Ripple	0.1 dB	0.1 dB
Stopband Attenuation	80 dB	80 dB

5.3.3 Gravity Filter

Due to the limitation of the IMUs we are using, we do not have gravity acceleration in our raw signals. We have to add gravitational acceleration to the measured acceleration to accurately describe the acceleration caused by patient movements.

First, we use a Mahony filter (algorithm 1) to convert acceleration, angular velocity, and orientation with respect to the earth's magnetic field into orientation estimated represented in quaternion coordinates. We then apply these calculated coordinates to a rotational matrix (equation 5.2) to convert the gravitational acceleration vector into the IMU's frame of reference.

$$g(x) = 2 * (i * k + j * r) * G \tag{5.2a}$$

$$g(y) = 2 * (j * k - i * r) * G \tag{5.2b}$$

$$g(z) = (r^2 - i^2 - j^2 + k^2) * G \tag{5.2c}$$

where

$$G = -9.81 m/(s^2) \tag{5.2d}$$

Combining the measured acceleration and gravity acceleration, we obtain the actual acceleration as follows:

$$a_{true}(x, y, z) = a_{measured}(x, y, z) + g(x, y, z) \tag{5.3}$$

Algorithm 1: Pseudocode for the Mahony Filter

Data: $Q \longleftarrow < Q_r, Q_i, Q_j, Q_k >$, $A \longleftarrow < A_i, A_j, A_k >$, $G \longleftarrow < G_i, G_j, G_k >$, $M \longleftarrow < M_i, M_j, M_k >$, Δt, $IntegralError \longleftarrow < IntegralError_i, IntegralError_j, IntegralError_k >$, K_i, K_p

Result: r, i, j, k

$error = \longleftarrow normalizeAcceleration$(A);

if $error == true$ **then**

  return;

end

$error = \longleftarrow normalizeMagnetometer$(M);

if $error == true$ **then**

  return;

end

$H_i, H_j, B_i, B_j \longleftarrow findReferenceDirection()$;

$estimateGravityDirection$(A);

$estimateMagneticFieldDirection$(M);

$Error_i, Error_j, Error_k$ = calculateError(A, M);

if $K_i > 0$ **then**

  $IntegralError_i \longleftarrow IntegralError_i + Error_i$;

  $IntegralError_j \longleftarrow IntegralError_j + Error_j$;

  $IntegralError_k \longleftarrow IntegralError_k + Error_k$;

else

  $IntegralError_i \longleftarrow 0$;

  $IntegralError_j \longleftarrow 0$;

  $IntegralError_k \longleftarrow 0$;

end

$Feedback_i \longleftarrow Feedback_i + K_p \times Error_i + K_i \times IntegralError_i$;

$Feedback_j \longleftarrow Feedback_j + K_p \times Error_j + K_i \times IntegralError_j$;

$Feedback_k \longleftarrow Feedback_k + K_p \times Error_k + K_i \times IntegralError_k$;

$Q \longleftarrow integrateRateOfChange$(Q);

$norm = normalizeQuaternion$(Q);

$r, i, j, k \longleftarrow Q_r \times norm, Q_i \times norm, Q_j \times norm, Q_k \times norm$;

return r, i, j, k;

5.3.4 Hampel Filter

We use a Hampel filter [14] to remove the random spikes in sEMG recordings likely produced from poor contact between the patient's skin and the sEMG electrodes. The moving window size is chosen to be 17, that is, we replace the outliers by the median of the neighbouring 8 data points to the left and to the right respectively (equation 5.4).

$$m_k = \text{median}\{x_{k-8}, \ldots, x_k, \ldots, x_{k+8}\} \tag{5.4}$$

5.4 Implementation

In this section, we will discuss how we implement the above filters in MATLAB.

5.4.1 Digital Filters

We used signal processing toolbox to generate the code for the low-pass filter and bandpass filter. The output of this code (MatLab Code Low-pass and Bandpass Filter) is a series of filter coefficients that could be convoluted to function as digital filters. We import these coefficients into our server for further usage.

In our server, we use 'conv' function from 'numpy' library in python to perform low-pass filter and bandpass filter on the raw signal. the detailed code is shown in Server Code - LowPassFilter.py and Server Code - BandPassFilter.py

5.4.2 Gravity Filter

We first designed the Mahony filter in python base on algorithm 1). Then we use the quaternion data genereated by Mahony filter to calculate the rotation matrix for gravitational acceleration by MATLAB (MatLab Code Gravity Filter). After we tested its performance, we implemented this filter in our server in python as shown in Server Code - GravityFilter.py

5.4.3 Hampel Filter

We first use the 'hampel' built-in function in MATLAB to test if this filter works for our sEMG signal. We then changed the window size to obtain an optimized parameter for best performance. Then we implement this filter to server in python with 'pandas' library (Server Code - HampelFilter.py).

5.5 Supporting Analysis and Testing

In this section, we will discuss some of the prototyping results we obtained by methods we discussed above. We tested all filters to check if they could meet the requirements. In general, our filters perform well and could generate desired results.

5.5.1 Tremor Model

Our model tremor has a peak frequency at 6.8 HZ as shown in figure 5.1, which falls into the PD tremor range of 4 HZ to 9 HZ. However, we did not find any clear second or higher order harmonics, which are commonly shown in PD tremors [13]. It is difficult to mimic the harmonics by healthy patients. We decided to continue using this model tremor for further analysis since it is sufficiently accurate for the scope of this research.

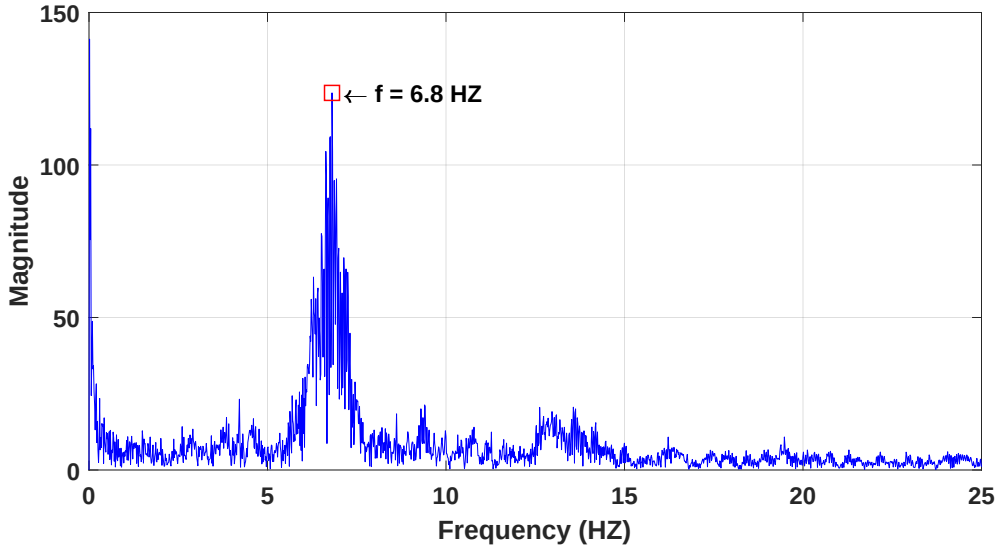


Figure 5.1: We performed FFT analysis on our tremor model. We obtained the peak frequency at 6.8 HZ.

5.5.2 Low-pass Filter

We used the low-pass filter designed by the method we described above and tested with a 300 seconds signal. Figure 5.2 shows that our filtered signal is less noisy than the raw signal. We then used FFT frequency analysis to test if the high frequency noise is efficiently cleared our by the filter. Figure 5.3 shows that all high frequency components are eliminated by the low-pass filter.

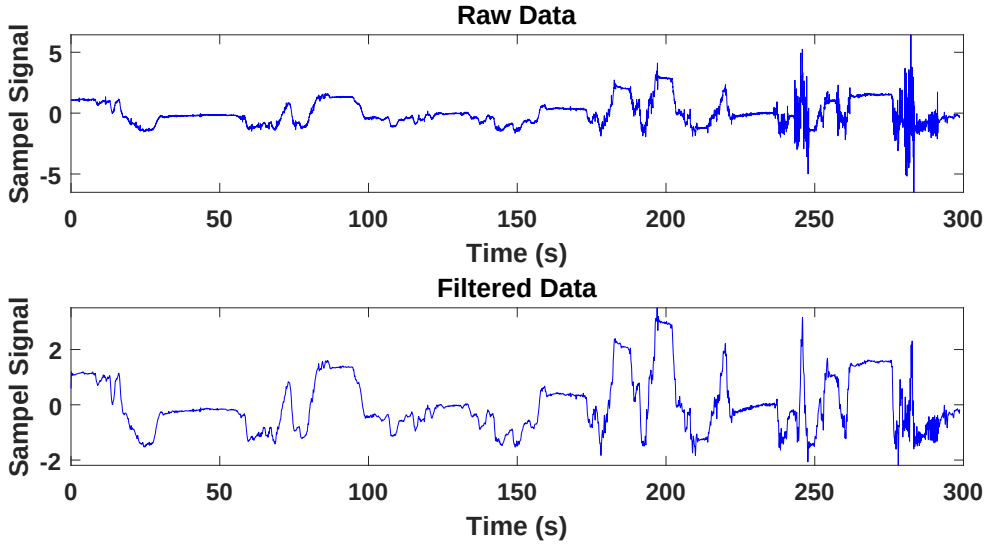


Figure 5.2: We used low-pass filter to analyze a sample signal. The filtered signal is less noisy than the raw signal.

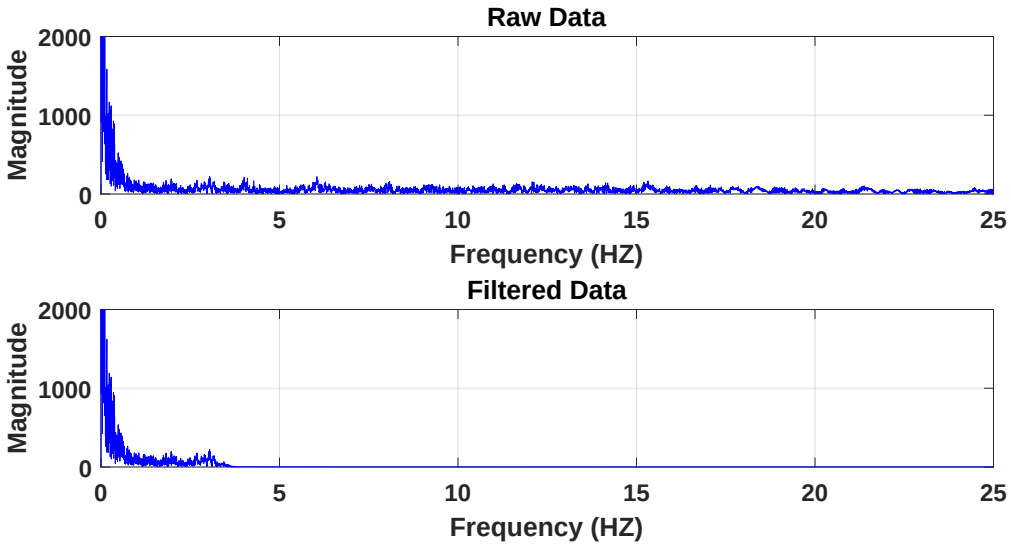


Figure 5.3: We use FFT to analyze the filter performance. The high frequency components are eliminated by the low-pass filter.

5.5.3 Bandpass Filter

We used the bandpass filter designed by the method we described above and tested with a 300 seconds signal. Figure 5.2 shows that our filtered signal has no directional movement except the high frequency tremor. We then used FFT frequency analysis to test if only desired frequency component is left in signal. Figure 5.3 shows that we have only signals in frequency of 3 HZ to 7 HZ left in filtered signal.

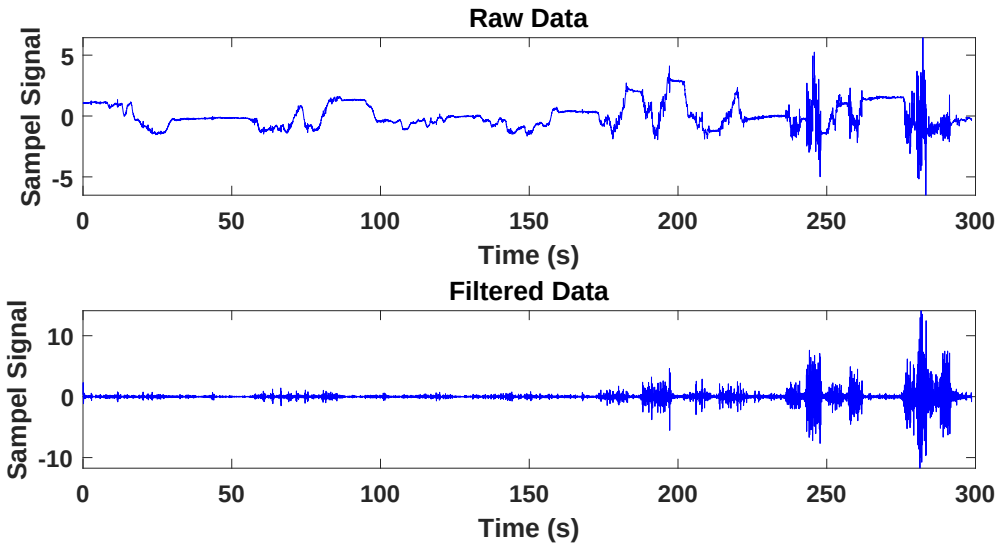


Figure 5.4: We used bandpass filter to analyze a sample signal. The filtered signal has no directional movement except the high frequency tremor.

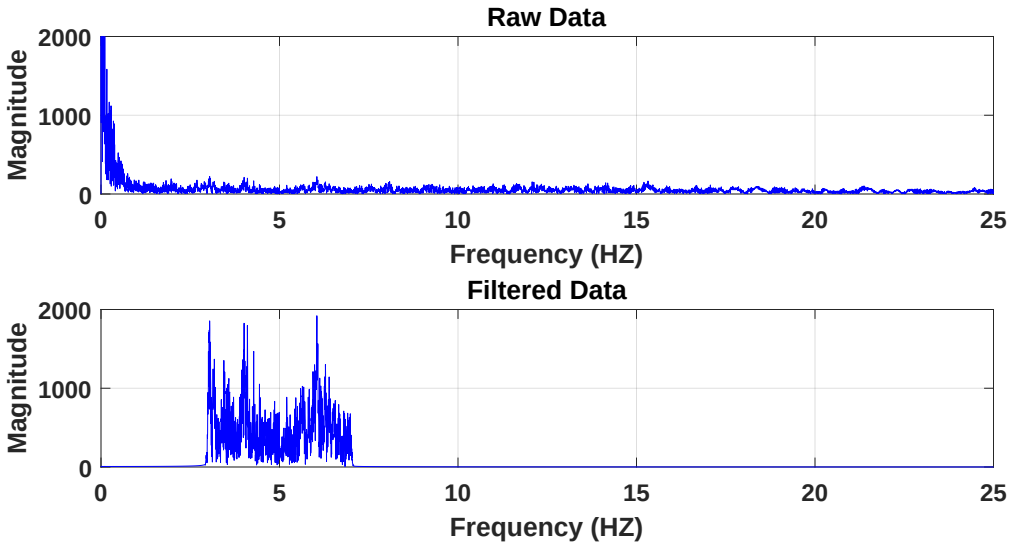


Figure 5.5: We use FFT to analyze the filter performance. Only signals in frequency of 3 HZ to 7 HZ left in filtered signal.

5.5.4 Gravity Filter

The performance of gravity filter is tested by putting our sensors on stable on a table and rotating the sensors without moving their gravity center. With the gravity filter, the accumulated acceleration should be close to 0. We used 400 seconds raw data for testing and the result is plotted in figure 5.6. We clearly showed that our gravity filter can rectify raw data. The position of our sensors were altered between 100 s to 150 s and the gravity filter performed well with different sensor position.

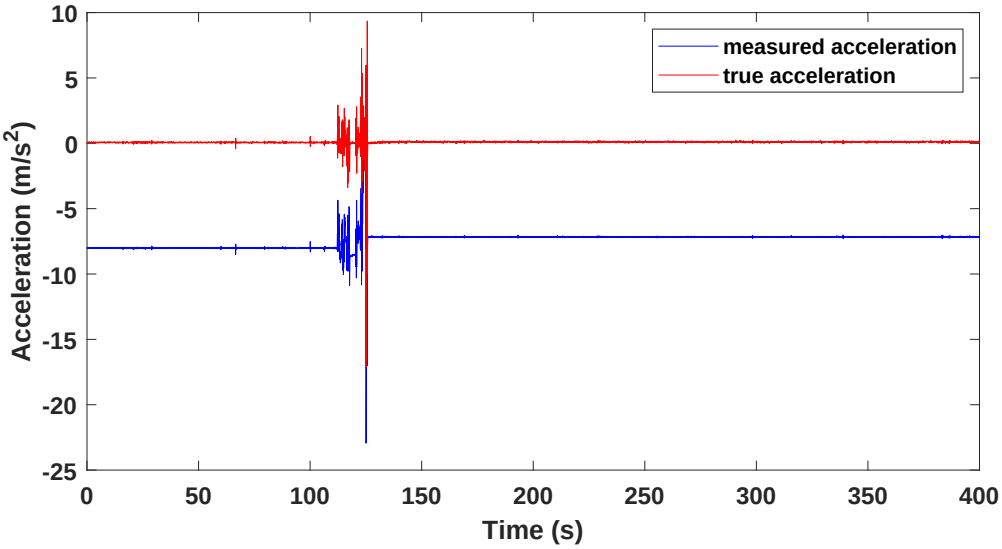


Figure 5.6: Gravity Filter

5.5.5 Hampel Filter

Since our hampel filter is specifically designed for the sEMG signal, we use a short sEMG signal to test its performance. Figure 5.7 shows that our hampel filter could remove most of spikes caused by skin hair and sEMG sensor.

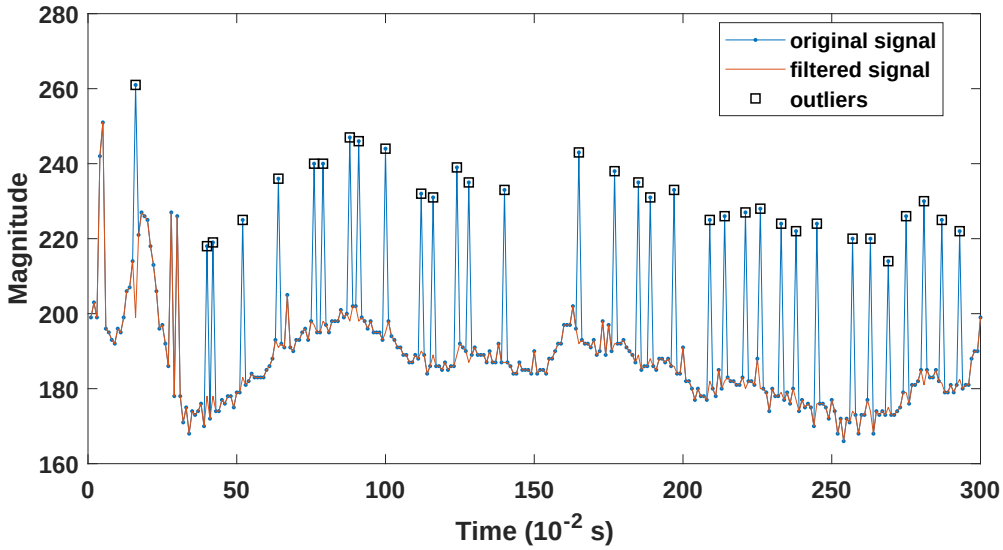


Figure 5.7: We tested our hampel filter on a sample sEMG signal. It removed most outliers.

5.5.6 Filter Efficiency

The efficiency of filters are determined by time consumption of filtration. It costs less than 0.2 seconds to filter the entire 1 hour data, which means all filters are fast enough for real-time analysis implementation.

Chapter 6

Machine Learning

Our system uses a machine learning algorithm in order to evaluate the patient's condition without the need for choreographed movements. The role of the machine learning subsystem is to determine significant sampling moments during a patients daily activities and pass the relevant information needed to evaluate a UPDRS score. We first gather the necessary data needed to train the model, then we train the model using a variety of different Machine Learning methods and test the results to see which model gives us the best results.

Overall, we have explored seven Machine Learning Models for our project. The first two models were classification models, which are similar to regression models in the sense that they are both forms of supervised learning, however, classification is used to predict a discrete output while regression is used to predict a continuous output. Our results with these two models were great, however, we realized that the model contained a large bias, which made our results less trustworthy. We then tried Linear Regression and Ridge Regression, not only in closed form, but also with gradient descent. These models were more accurate, however, the Linear Regression models had a tendency to overfit the training data, and the Ridge Regression models had less bias but greater variance in predictions, which also made our results less trustworthy. Finally we implemented Logistic Regression, which had a much better result than our previous models due to its higher level of complexity.

6.1 Functional Requirements

- The Machine Learning algorithm must be able to recognize the movements associated with assessing Parkinson Disease tremors based on the following two examinations taken from the the MDS-UPDRS Motor Examination Section [3].
  - UPDRS 3.4 Finger Taps
  - UPDRS 3.5 Hand Movements

- The system will assess these two tests by analyzing the patient's regular daily actions in the home or identifying choreographed test autonomously.

6.2 Non-functional Requirements

- The system will require the minimum effort on the part of both the patient and doctor for setup, running, and maintenance.
- The system must evaluate raw data and produce UPDRS scores by the end of the recommended period which the system is utilized by the patients.

6.3 Logistic Regression using Gradient Descent

$$0 \leq h_\theta(x) \leq 1$$

Figure 6.1: Our desired hypothesis lies between 0 and 1 because we want to evaluate/predict whether an action occurred or not. Zero means that the model predicts the action did not occur and one means that the model predicts the action did occur.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Figure 6.2: In order to map a continuous value between zero and one, we must use a function with an 'S' shape and boundaries at zero and one. There are a few options for this function, but we decided to use the Sigmoid function since it is the most commonly used.

$$h_\theta(x) = Sigmoid(\theta^T x)$$

Figure 6.3: Our hypothesis function takes an input of features x, and a vector of weights theta, performs a matrix multiplication to combine the inputs and weights, then the Sigmoid function is applied to the resulting matrix

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} log h_\theta(x^{(i)}) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))]$$

Figure 6.4: Now we must derive our cost function. This function is often referred to as a loss function.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$$

Figure 6.5: Next we take the derivative of our cost function with respect to theta in order to minimize our cost.

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

Figure 6.6: Using the minimized cost function, we can begin our gradient descent towards an optimized solution by updating the weights at every iteration.

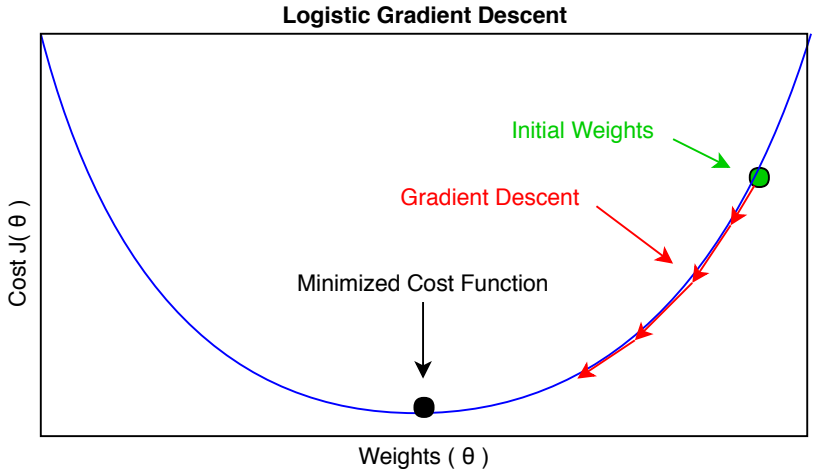


Figure 6.7: This is a generalized graph to help conceptualize the way gradient descent works. We begin by initializing random weights and then following the gradient by updating the weights until we have a minimized cost function

6.4 Datasets and Models

Since there was no data to begin with, we built a device that accurately gathered data in order to use that data to train a model to make good predictions about newfound data. We first performed many variations of non-choreographed finger tap and hand grasp actions used in the UPDRS examination. Then we performed many variations of the interruption action that occurs when one of the two previous actions are attempted but not completed. This resulted in 4 categories: 1 Finger Taps, 2 Hand Grasps, 3 Finger Tap Interruptions, and 4 Hand Grasp Interruptions. These four actions seemed to take place in a range of frequencies, so we decided to gather data at 4 different frequencies: 0.33Hz, 1Hz, 2Hz, and 3Hz. The product of these categories and frequencies are 16 models that correspond to every possible combination of one of 4 categories with one of 4 frequencies:

- **1.1** Finger Taps at 0.33 Hz with a sampling period of 300 data instances
- **1.2** Hand Grasps at 0.33 Hz with a sampling period of 300 data instances
- **1.3** Finger Tap Interruptions at 0.33 Hz with a sampling period of 300 data instances

- **1.4** Hand Grasp Interruption at 0.33 Hz with a sampling period of 300 data instances
- **2.1** Finger Taps at 1 Hz with a sampling period of 100 data instances
- **2.2** Hand Grasps at 1 Hz with a sampling period of 100 data instances
- **2.3** Finger Tap Interruptions at 1 Hz with a sampling period of 100 data instances
- **2.4** Hand Grasp Interruption at 1 Hz with a sampling period of 100 data instances
- **3.1** Finger Taps at 2 Hz with a sampling period of 50 data instances
- **3.2** Hand Grasps at 2 Hz with a sampling period of 50 data instances
- **3.3** Finger Tap Interruptions at 2 Hz with a sampling period of 50 data instances
- **3.4** Hand Grasp Interruption at 2 Hz with a sampling period of 50 data instances
- **4.1** Finger Taps at 3 Hz with a sampling period of 33 data instances
- **4.2** Hand Grasps at 3 Hz with a sampling period of 33 data instances
- **4.3** Finger Tap Interruptions at 3 Hz with a sampling period of 33 data instances
- **4.4** Hand Grasp Interruption at 3 Hz with a sampling period of 33 data instances

6.5 Implementation

6.6 Testing

We trained each of these models to recognize their respective patterns of movement hidden in a patient's daily activities. Quantifying these four actions in a data session allowed us to score the two UPDRS tests described in the requirements by evaluating the ratio between completed actions and interruptions. However, machine learning models only function as intended when enough quality data is available to train on. So in order to maximize the quantity of our data, we developed a form of data manipulation that takes a single session of gathered data and multiplies the number of instances by the sampling rate, thus resulting in a much larger dataset that still contains unique instances.

In order to test the quality of our Machine Learning model, we train using a training dataset, then we test its prediction accuracy using a different dataset, and compare the predictions with the actual values in order to determine the error rate of our model. Since we were unable to gather patient data, we had to replicate the tremor movement as best we could, and as a group of three, we trained off of two of our data sessions and tests on the third so that the trained model would be tested on data it had never seen before.

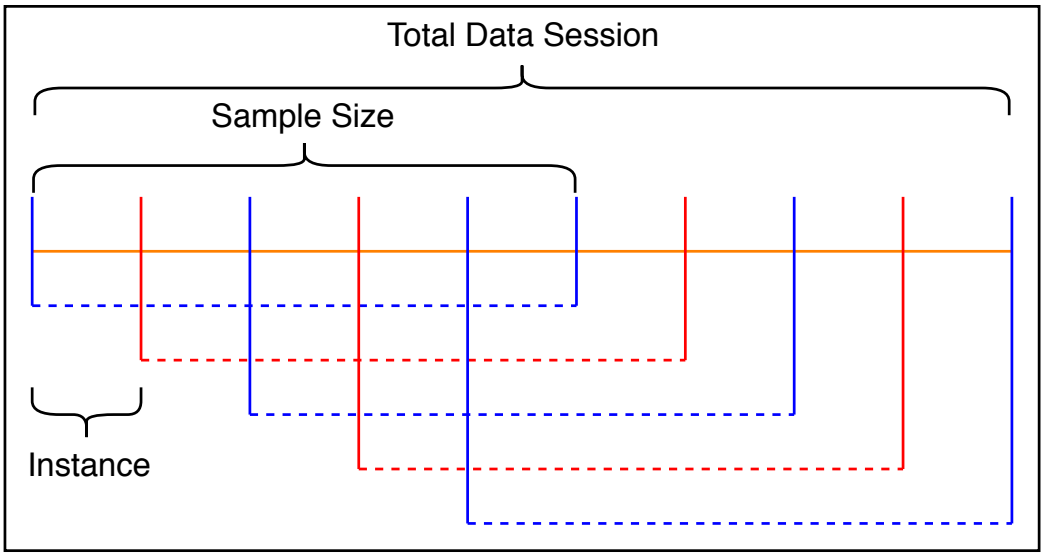


Figure 6.8: We gathered 480 data samples and from that we generated 57,963 unique data samples.

$$16(models) \times 3(subjects/model) \times 10(samples/subject) = 480(datasamples)$$

$$480/4 = 120$$

$$120 \times 32 = 3,840$$

$$120 \times 49 = 5,880$$

$$120 \times 99 = 11,880$$

$$120 \times 299 = 35,880$$

$$3,840 + 5,880 + 11,880 + 35,880 = 57,480$$

$$57,480 + 33 + 50 + 100 + 300 = 57,963$$

Keep in mind, there are some cases where the results will be misleading, that is, if the model has overfit the training data (this means that the model contains a large bias, and has possibly learned to recognize noise as opposed to data). Therefore, we are using cross validation, feature removal, and regularization to ensure that it is not overfitting the data. With these safety measures in place, we managed to reach an average accuracy upwards of 90% for these two tests. Admittedly, the system would occasionally mistake a 1Hz action with a 2Hz action, however, even in these test cases it would still quantify the action with minimum of 80% accuracy.

As we explore these Machine Learning models, we learn more about the information hidden within our data. We firmly believe that our idea's feasibility has been verified in the sense that a Machine Learning algorithm is capable of performing the specified evaluations of the UPDRS test, however, we also realize the shortcomings of our design. For instance, although we have managed to closely replicate a tremor movement, our data is still limited at the moment because we lack data from actual tremor patients. With this in mind, we

redesigned our model to make the most out of the data we could gather from ourselves while simultaneously recognizing the potential of real patient data to further improve the model.

Chapter 7

Scoring Subsystem

7.1 Introduction

The last subsystem we designed is the scoring subsystem. We utilize the raw data collected from our device, digital filters, and the machine learning algorithm to generate a estimated UPDRS score (table 7.1). We perform six tests from UPDRS on finger and hand tremor, with the expectation of implement more tests in the future. Once all tests are complete, a scoring report is generated as shown on the second page of the Appendix.

Table 7.1: Data, Filters, Machine Learning Used in Score Subsystem

	Finger Taps	Hand Movement	Postural Tremor	Kinetic Tremor	Resting Tremor	Constancy of Tremor
sEMG	✓	✓	✓	✓	✓	✓
Accleration	✓	✓	✓	✓	✓	✓
Angular Velocity	✓	✓	✓	✓	✓	✓
Low-pass Filter	✓	✓	✓	✓	✓	
Bandpass Filter						✓
Gravity Filter	✓	✓	✓	✓	✓	✓
Hampel Filter	✓	✓	✓	✓	✓	✓
Machine Learning	✓	✓				

7.2 Options

We picked six tests from the Motor section of UPDRS with focus on finger and hand tremor. The first two tests are Finger Taps and Hand Movement, which requires pattern recognition with machine learning. The rest four tests are Postural Tremor of Hands, Kinetic Tremor of Hands, Rest Tremor Amplitude, and Constancy of Tremor. These four tests rely on bio-mechanics and frequency analysis.

We picked these six tests based on the limitation of our device. Our sensors are placed only on fingers

and the dorsum of palm. We try to balance between the size of entire design and comfort of wearing. We believe that these six tests are representative of hand tremor and could prove the concept that we can estimate a UPDRS score with non-choreographic actions and without professionals presence.

7.3 Finger Taps and Hand Movement

The methods we used for Finger Taps and Hand Movement are similar, therefore, we describe both methods together. The original UPDRS Finger Taps and Hand Movement tests consists three parts: interruption counts, movement slowing, and amplitude decrements. In our design, we only implement the interruption counts to show the capability of machine learning in determination of UPDRS score.

First we run the new patient data through every one of our machine learning models in order to determine what frequency should be used to evaluate the data. Once we have a frequency, we then count all the relevant actions and attempted actions. Finally, we evaluate the score by taking the ratio of a specific action to the interrupted attempts of that same action during the span of the sampling session. The value of the ratio determines the score they are assigned as specified in the algorithm 2.

Algorithm 2: Scoring Algorithm for Finger and Hand Movement

Data: action_count, interrupt_count

Result: score

if *0≤action_count and interrupt_count* == 0 **then**

    score ← 0;

    return *score*;

end

if *1≤action_count and 1≤interrupt_count and 2≥interrupt_count* **then**

    score ← 1;

    return *score*;

end

if *3≤action_count and 3≤interrupt_count and 5≥interrupt_count* **then**

    score ← 2;

    return *score*;

end

if *5≤action_count and 5≤interrupt_count* **then**

    score ← 3;

    return *score*;

end

7.4 Tremor Amplitude and Constancy

In this section, we perform the rest four UPDRS tests: Postural Tremor of Hands, Kinetic Tremor of Hands, Rest Tremor Amplitude, and Constancy of Tremor. These four tests require us to first determine when the patient has tremor and analyze the tremor amplitude by the features we generated from signal analysis subsystem.

The first step is to determine when a postural tremor, a kinetic tremor, or a resting tremor happens. We first cut the entire signal into 100 pieces with equal length. The number of pieces is optimized with algorithm speed and accuracy. Then for each pieces, we analyze the filtered velocity and sEMG reading to determine the action. Postural tremor happens when the muscle is contracted while the velocity is close to 0 in all x, y, and z direction. Kinetic tremor happens when when the muscle is contracted while the velocity in y-direction is higher than 0, which means the hand is reaching out. Resting tremor is evaluated when a certain muscle is relaxed and all three directions of velocity is zero. The criteria is summarized in table 7.2.

Table 7.2: Criteria for Three Types of Tremor

	Postural Tremor	Kinetic Tremor	Rest Tremor
sEMG	High	High	Low
Velocity	Low	High	Low

The second step is to calculate the tremor amplitude corresponding the those three types of tremor. With bio-mechanics analysis, we find that the amplitude could be calculated by the radius of hand and roll angle in Euler angles. The amplitude is calculated as follows:

$$Amplitude = 2 * R_{hand} * tan((\Delta Roll_{hand})_{max}); \tag{7.1}$$

Based on the amplitude, we assign score for these three tremors.

The last step is to calculate constancy of tremor. We use similar strategy to cut the signals to 100 pieces. We analyze the signals treated by bandpass filter. If the majority of the signal exceeds the tremor limit we set, we count this piece as a tremor piece. The constancy of tremor is then calculated as follows:

$$Constancy = \frac{TremorPiece}{100} \tag{7.2}$$

Based on the constancy, we assign score for this last test.

We first implement these four tests in MATLAB as shown in MatLab Code Score in Appendix. Once it functions correctly, we translate this part into python and put it in Server Code - Score.py in Appendix.

7.5 Reporting

The reporting functionality is intended to be useful for health care professionals and researchers. The reporting module extracts the scoring data and formats it in the form of a PDF report. As seen in the sample report in the appendices, the user will receive the patient's automatic UPDRS scores as well as graphs of the physical data. In the future, more graphs will potentially be added to improve the usefulness of the report.

Chapter 8

System Integration and Testing

This section describes subsystem tests to verify that the tests match the requirements of the system. Each test will provide a description of the procedure as well as the subsystem requirements that it covers.

8.1 Device

Testing for the device subsystem have focused so far on the validation of the device's performance. Such measures include graceful reactions while in failure modes, the capability to measure the correct amount of information, and the capability to accurately measure information.

8.1.1 Failure Testing

We also perform failure testing on the device to evaluate it's ability to respond to failure states, such as the poor initialization of hardware or an exception error which prevents the execution of code. The utmost ideal response of a device to a failure is to automatically reset the device and attempt to automatically fix the problem. Since our device does not have such a feature yet, we look for graceful failures. A graceful failure in the context of this project means that the device logs the error in its log file in permanent storage before entering a kill state, which signals the user to reset the device.

Three different tests were performed to evaluate if the test can correctly handle failures. The first test allowed for the device to run normally. The first log in figure 8.1 exemplifies a successful performance, which utilizes each hardware feature. For the next test, several wires from an IMU were intentionally pulled out before the device had started its initialization process. The device reacts accordingly as seen in the second log file of figure 8.1 by notifying the user that one of the IMUs has failed to initialize and providing a log code. The device then after enters the kill state in which the device flashes its light to notify the user that there is a problem. For the final test, the rate of the consuming thread was set slower than the producer to intentionally cause the circular sampling buffer to overflow. The third log file in figure 8.1 shows that as soon as the buffer

overflows, the device triggers an exception and gracefully shuts the device down for maintenance.

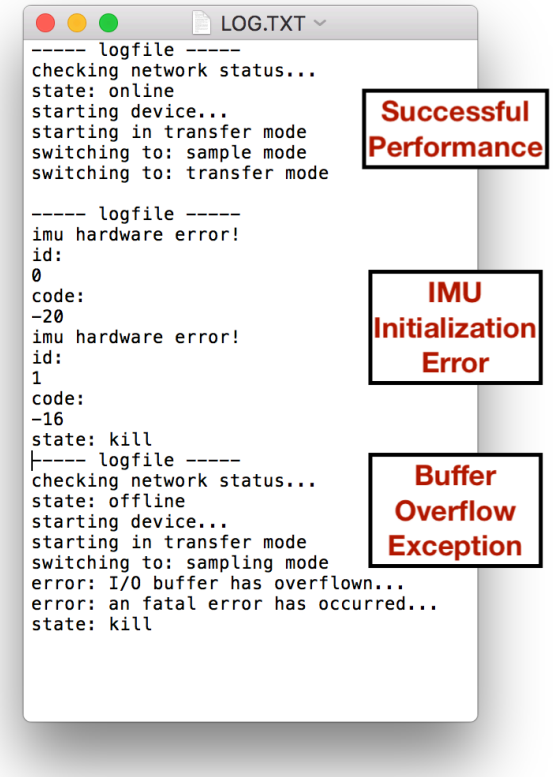


Figure 8.1: Hardware and Software Failure Testing

8.1.2 Persistance of Measurements

This test verifies that the amount of time spent monitoring the patient is close to the amount of information that the device is supposed to collect. To ensure this, the device was sampled for several periods of time and compared with the expected results. Table 8.1 shows each test performed, the expected result, and the actual result when sampling for these times.

Since each sample is buffered in the device before it is stored, when the device is shut down it is likely for a few not to be recorded. While this is true, table 8.1 shows that at least ninety nine percent of all the required data still is recorded. This implies that the device subsystem has proven itself to be reliable for recording data for the required amount of time so that UPDRS scoring can be done.

8.1.3 Energy Efficiency

The energy efficiency of the device determines for how long a patient can collect and transfer data with this device for UPDRS scoring. The goal of this test was to collect feedback of the actual lifetime of the device before applying power consumption behaviors. The glove was powered on the test bench with all

Table 8.1: Test Results

Time to Sample	Expected Sample Count	Actual Sample Count	Sample Count Ratio
1 minute	6,000	6,087	1.0145
5 minutes	30,000	30,090	1.003
30 minutes	180,000	179,899	0.9994388889
1 hour	360,000	359,343	0.998175

hardware actively working, to determine how close the actual battery life was to the calculated battery life according to equation 3.3. The results show that the device lasted approximately 15.16 hours of constant normal usage before losing its capability to power any one component.

8.1.4 Measuring Accuracy

The quality of the low-cost inertial measurement units has been evaluated through the comparison with the Physilog5 sensor, which has been recommended for medical sensing and evaluation. We would like to compare the quality of the MPU9250 to the quality of the Physilog5 sensor to justify its potential for research and medical accuracy for a low-cost platform. To compare both sensors, the tester wears the glove and additionally places the Physilog5 on top of the IMU located on the dorsum of the palm. Both devices should be oriented such that there axes are similar. The patient then performs five minutes worth of fast and slow movements to evaluate sampling ability. Both data sets are then subtracted from each other to find the difference. Figure 8.2, presents box and whiskers plots between each axis of acceleration compared. The medians for each plot are close to zero and the upper and lower quartiles are fairly thin, which signifies that the data is approximately similar. Unfortunately, for each plot there does a range of large outliers, clearly marking the expected difference in quality between the two sensors. While the Physilog5 sensor measures far more accurately, we can potentially filter out these outliers to make the sensor more stable for medical monitoring.

8.2 Processing Time

The filter performance was additionally tested as an entire pipeline. Figure 8.4 shows the linear increase in the pipeline's processing time as we fed different sized sets of raw data onto the server. Linear increase proportional to load increase is promising in terms of computational scaling as we continue to develop this platform for the future.

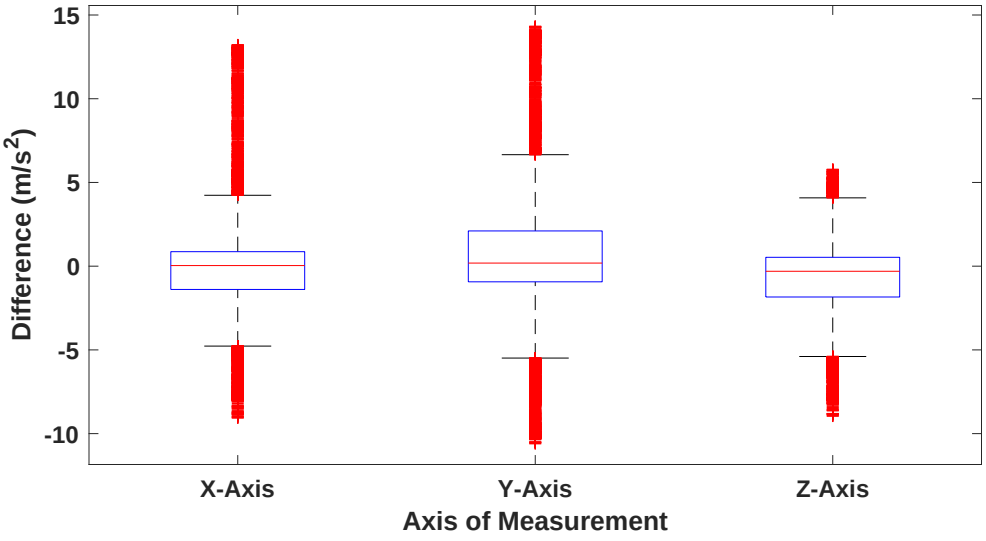


Figure 8.2: Acceleration Quality Comparison

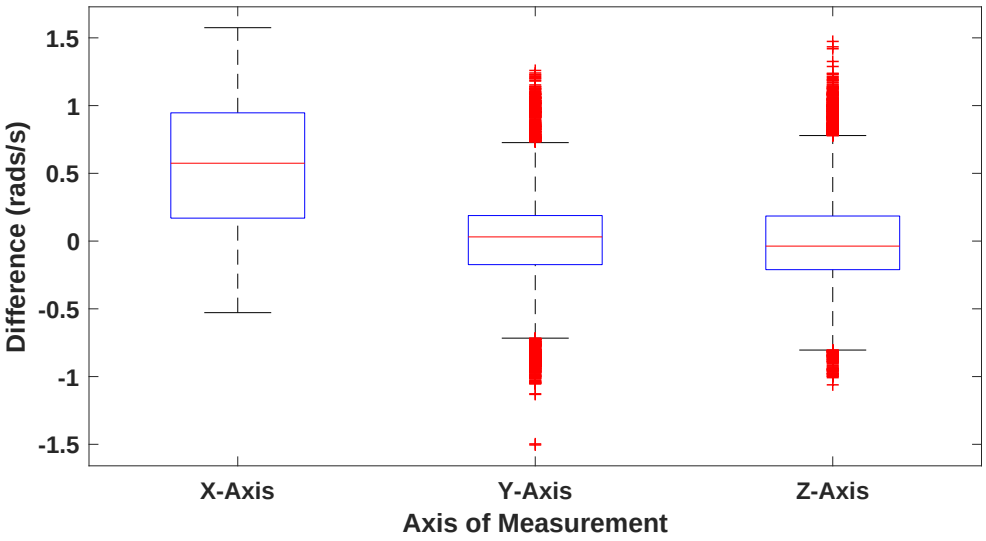


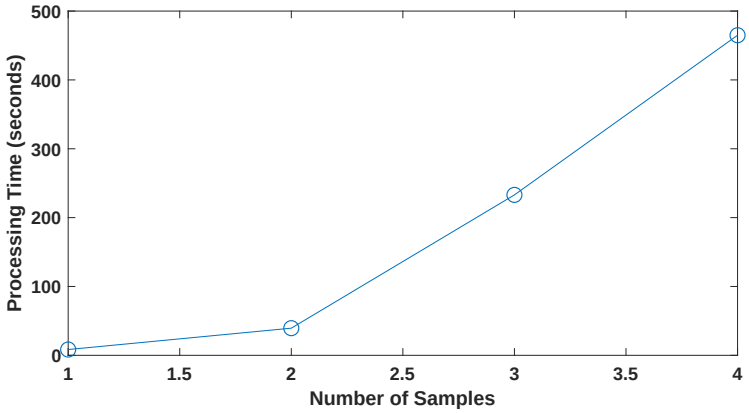
Figure 8.3: Gyroscope Quality Comparison

8.3 Filters Performance

Several different filters are presented in Chapter 5. We want to test performance of our filters in the whole system.

The performance of low-pass filter is determined by the improvement it can provide to position determination. We used both raw data and filtered data to train our machine learning algorithm and compared the performance of the model that machine learning generated. We compared sensitivity and specificity of all 16 models and plot the result in Fig. 8.5. The low-pass filter can improve both sensitivity and specificity of the

Figure 8.4: Pipeline Processing Time of Raw Data



machine learning algorithm by keeping both rate above 0.4. Given the fact that we trained our models with only ten samples each, we still have potential to improve our result.

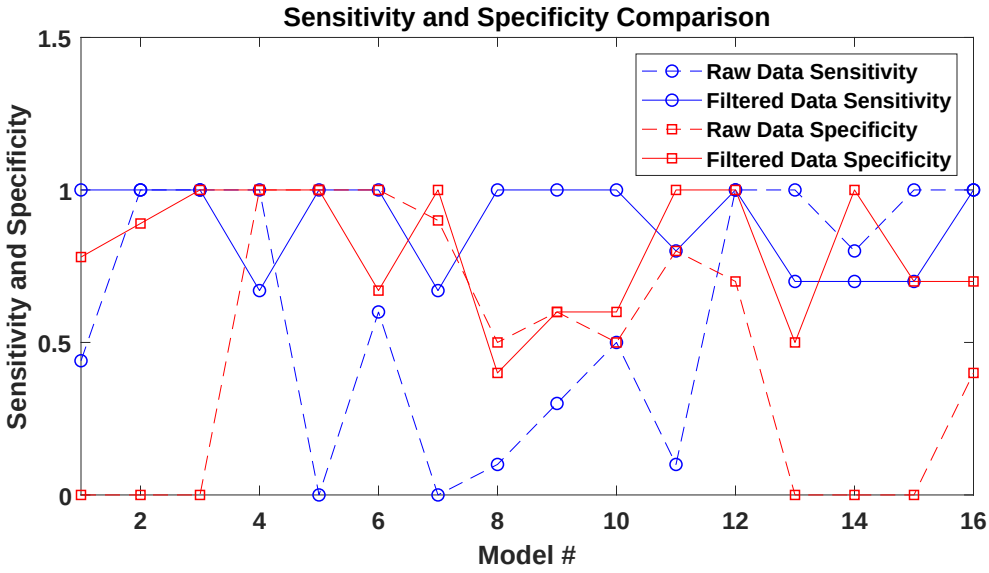


Figure 8.5: Sensitivity and Specificity Comparison

8.4 Machine Learning

This section describes the testing process for the machine learning scripts used to determine important actions that the patient performs that could be useful for UPDRS scoring.

One major concern with training and testing is the alignment of features and weights. We began gathering data from our first prototype so that we could attempt to train a model off the data and it worked wonderfully. However, by the time we had trained our model, we had also upgraded our device, and the second prototype

had expanded the range of features being retrieved so new data was not matching the dimensions of the model correctly. This is a problem because the model is trained by establishing a set of weights that help make a prediction when properly applied to our hypothesis function, but if the weights are not aligned with the features, the result can break the model. This error led to numerous tests on the dimensionality of weights against input features in order to guarantee that they matched. We even had to regather all of our data so that our model could handle the dimensionality of the features gathered in the latest prototype.

Chapter 9

Costing Analysis

This section describes all of the costs attributed to the project. Costs can take several different forms including: money, time, and space. The device and system monetary costs were small per unit, because the entire system was designed using low-cost parts. This project, however, was more costly in terms of time and space.

9.1 Time Costs

There was a lot of time put into this project for each different stages. Of these, the most pronounced time costs included: meeting costs, testing costs, development costs, and research costs.

An approximation of these times can be seen in the appendix figure 3, which provides an estimation of the project schedule throughout the year. On a weekly basis, the team met three times a week on Monday, Wednesday, and Sunday. The time spent on meetings per week was on average seven hours, with expected work time outside of the meeting time. That said, each member of the team spent on average over ten hours each week working on this project.

Testing costs consisted of a substantial portion of our time during the development phase. While we do not have a specific measurement as to how much time was spent, a majority of our meeting times consisted integration testing of the system. This integration testing was done both to confirm the functionality of the system and to collect data for the machine learning and signal processing subsystems. As an estimate, testing consumed roughly four out of the seven hours spent each week doing team meetings.

Research costs are the time we devoted to read scientific publications to learn knowledge for developing this project. During summer quarter and fall quarter, we read over 50 papers and patterns, from which we learned a lot of knowledge in Parkinson's Disease and the state-of-art design for designing different subsystems. The research time has benefited not only for this project but also for us as professional engineers.

9.2 Monetary Costs

System development costs were fairly inexpensive. Table 9.1 shows the estimated costs of materials to develop the wearable device, which in comparison to other sensing technologies is inexpensive. As a result, we were able to quickly purchase materials in bulk and develop multiple prototypes of the wearable device for development and refine our design more quickly.

Table 9.1: Wearable Device Cost

Device	**Unit Cost**	**Quantity**	**Cost**
Teensy 3.6	$29	1	$29
XBee S1	$25	2	$25
MPU 9250	$15	4	$60
Myoware EMG	$38	1	$38
		Total Cost:	~$152

The cost for the entire system would consist of the cost of the wearable as well as the cost to purchase the server. Although we used a laptop for development, the system could also just as easily use a cheap microprocessor, which relies on cloud computing for scoring. Even when using cloud services the cost would be fairly low.

Part of the goal of designing this platform was to ensure that the system was capable of running using low-cost products, which we hope will lower the barrier of entry for researchers attempting to design systems for automatic UPDRS scoring. If we continue to pursue this project, additional costs will be added in order for us to collect legitimate Parkinson's Disease data from hospitals and observe UPDRS scoring.

9.3 Space Costs

This project was fairly space-efficient, since all of the electronics were small and able to be carried in small boxes. The development space required only a power outlet powerful enough to charge two laptops, and a table on which the electronics can be placed. The most difficult aspect was finding a place to work where there is plenty of open space to organize electronics and tools to weld.

Chapter 10

Business Plan

The volatility and progression of PD symptoms has inspired the health care community to introduce home monitoring for PD patients. One recent review explains the significance of using wearable technology in the health care field and its potential impacts to controlling tremors [12]. In this report tremors have also been cited as one of the most disrupting symptoms in one's daily life. Through the accessibility of wearable technologies, we could potentially improve our responses to Parkinsonian symptom fluctuation.

Many researchers and teams now have their goals set of improving the UPDRS scoring process [5, 8, 9, 10, 15, 16, 17]. All of these researchers utilize the same types of equipment, including sEMGs and inertial measurement units to collect data. However each team utilizes this information differently, and developing insight on how to automate the UPDRS scoring process.

Several research teams sought to design systems focused on enabling patients to interact with health care professionals from their homes [16, 17]. These methods demonstrate successful proofs of concept for web services; they reduce the cost of travel and continue to provide professionals with enough data to make accurate evaluations about the patient. Furthermore, proposed methods such as these showed that patients can reliably perform the necessary actions at home in order to perform evaluations [7]. These systems, while they provide more quantitative data, still require a doctor to objectively make decisions, which may still affect the outcome of the treatment.

Many other teams focused on creating automated UPDRS scoring systems [5, 7, 8, 9, 10, 11, 15]. These automated UPDRS scoring systems also can be subdivided into systems that rely on guided human movement such as a UPDRS test example [7], or systems that monitored unguided movement [5, 8, 9, 10, 11, 15]. Several of the automatic scoring systems attempt to utilize machine learning to classify the different scoring metrics [5, 10]. Unfortunately both teams agree that machine learning, while promising, does not meet a high enough accuracy to perform as a suitable scoring method [5, 10]. Other teams mixed more numerical methods along with different classifiers, to explore the kinematic features correlated to specific UPDRS scores. One

team in models a promising method for automatic UPDRS scoring system for gait symptoms, which relies on the linear relationship between several kinematic features and the actual UPDRS scores [9]. Another team has designed a methodology that produces highly accurate estimates of tremors for UPDRS using an inertial measurement unit on a wristband [15].

Chapter 11

Engineering Standards and Realistic Constraints

In this chapter, we examine the implications of our system from different perspectives, and examine its impacts, or the lack of impact, in several different contexts.

11.1 Ethical

The most common ethical issues with regards to the meld of smarter embedded technologies in the medical field usually are closely related to privacy issues. Our device will be monitoring a patient's hand movement throughout the day, which can be concerning to a patient. From the Utilitarian ethics perspective, although the patient sacrifices the privacy of their movement, as long as they consider the treatment of their Parkinson's Disease symptoms as valuable, we believe that the UPDA system is ethically justifiable. There has also been a long debate of the ethics of using machine learning in the medical field. With the push of the UPDA system, we would be affirming that it is ethically justifiable to use machine learning in human treatment, which may upset some people.

11.2 Social

This technology aims to remove the cost of measuring Parkinson's Disease from a patient's day-to-day life. In that regard, this technology is even enabling patients to be more social in the community. However can be some social drawbacks to wearing this device in the community. By having the patient wear a device that is visible to others, the device could be used as a feature to identify someone as having Parkinson's Disease, which could affect the patient's social interaction with the community for better or for worse.

11.3 Political

As a wearable device in the medical field, we do not believe it will have any impact on politics. The one exception, however, is that our device could be involved in political discussion involving the internet of things into the medical field. Even if there were any politics surrounding wearable devices in the medical field, the UPDA device would very likely would not change any political thoughts, just only affected by political agendas.

11.4 Economic

Many research teams have attempted to design automated UPDRS scoring systems, but the UPDA could potentially be the first on to market. In the case that UPDA makes it to market, it will create a whole new category of medical device, which will likely be met with competitors now rushing to bring their products to market.

11.5 Health and Safety

We expect our system to focus on the maximization of health and safety. Our requirements and system are designed to improve the health of the patient with the smallest inconvenience. The product must be incredibly safe, and in order for it go get to market, it must gain FDA approval as a class one medical device. All these rules and regulations will help shape our system to ensure that it only improves the condition of a patient's PD symptoms and in no way worsens them or introduces any other health and safety issues.

11.6 Manufacturability

Our project can be manufactured differently depending on the purpose. All of the main electronic components for the prototype are low-cost and off-the-shelf, thus we hope that many other research teams will be able to reproduce our work effectively and for a low cost. The device can also be designed as a more serious product, which we intend to do as the product continues. As the system matures, the hardware may become more customized for the solution, increasing the barrier for entry for other groups attempting to replicate the product.

11.7 Sustainability

We believe that from an engineering standpoint, the system is flexible enough to a degree where it can continue to be developed by us or outside communities and continue to be relevant and helpful to people.

Unfortunately, there is no clear cure for Parkinson's disease, and so we hope that our system can continue to be as useful as we originally hoped it would be.

11.8 Environmental Impact

Wearable devices in particular have a set goal to last as long as possible. Ideally the device will be designed to battery life optimally, and extend the efficiency of one charge over a long period of time. By saving energy on the device, that means we are creating devices that require less charging. Less charging of devices means that less power has to be generated, which reduces the use of fossil fuels and nuclear generation. Inversely, if the patient had to constantly drive over to the hospital to perform UPDRS evaluations, it is highly likely that even more fossil fuel is used, worsening the atmosphere. In that light, and due to the sheer number of people with Parkinson's, we have the potential to reduce car traffic substantially.

11.9 Usability

Due to the difficulty of using devices as an advanced Parkinson's Disease patient, it was necessary for our device to be usable. The device is simply controlled with one button, and uses a simple light queue to tell the patient what is going on. The device even will notify the user if there is something wrong, so that the device can be easily corrected. The server side of the system is managed automatically to try and take as much effort off the patient and medical professionals as possible.

11.10 Lifelong learning

This project was an awesome tour of the world of medical device development and technology. From this project, we were inspired to take different classes outside of the normal curriculum to add new features to our project. We learned a lot about what it means to put together a medical device and how much effort and time it takes to thoroughly re-iterate the design process until it meets the requirements that we want for it. We also learned a substantial amount about machine learning in the medical device industry and its difficulties for implementation in a system. Overall, we all have been inspired to continue learning more about software tools and engineering in these fields and other fields that are closely related.

11.11 Compassion

Through our experiences talking with doctors, actual Parkinson's Disease patients, and even our friends who know someone with Parkinson's Disease, we have grown to realize how prevalent Parkinson's Disease

is around us. Additionally, through our research we have learned about how difficult the symptoms are to manage, and how much patients truly suffer each day.

Arguably, what makes this project truly real and what made us put so much effort into this project had to do with the potential impact it had on others lives. We have taken action at the chance to help relieve others suffering and we truly hope to always take action if there is anyone ever in need of an engineer.

Chapter 12

Conclusion

This paper presents the preliminary design and validation of a system for the automatic scoring of portions of the UPDRS based on a patient's regular movements. Our wearable medical device is capable of gathering over 50 features at a rate of 100Hz and has a battery life so great that if you were to use it for an hour each day of the week, you would only need to charge it about once a week. We have implemented a handful of signal filters and machine learning models that have proven capable of deciphering the patterns hidden in a patient's daily actions. We have also established that although a doctor is needed to prescribe proper medication, a doctor is not necessarily needed to perform this evaluation, therefore a product such as this would save patients a lot of money in the long-term.

To verify this system as a platform for future work we designed experiments to confirm its functionality to gather raw data and produce features through filters. We performed several experiments, either with the device alone or with our team as test subjects. Our current version of the glove is capable of supplying researchers and patients with continuous symptom monitoring at useful resolutions. The glove only requires the push of a button, which makes the device more accessible for patients to use.

The system architecture offloads the heavier processing onto the server, allowing for more powerful data processing than possible on board the device. The server also creates options for future services; doctors with permissions could request patient data remotely for simplified treatment adjustment.

We fully intend to continue this work further on the system as a whole; improving its functionality for health care professionals as well as the quality for scoring. We would also like to organize patient testing with a facility that is capable, to gather information from actual patients. This platform has demonstrated potential to become a foundation for future development for PD monitoring and automatic UPDRS scoring with uncoreographed daily movement.

Bibliography

[1] G. . Disease, I. Incidence, and P. Collaborators, "Global, regional, and national incidence, prevalence, and years lived with disability for 310 diseases and injuries, 19902015: a systematic analysis for the Global Burden of Disease Study 2015," *Lancet (London, England)*, vol. 388, pp. 1545–1602, Oct. 2016.

[2] K. Seppi, D. Weintraub, M. Coelho, S. PerezLloret, S. H. Fox, R. Katzenschlager, E.-M. Hametner, W. Poewe, O. Rascol, C. G. Goetz, and C. Sampaio, "The Movement Disorder Society Evidence-Based Medicine Review Update: Treatments for the non-motor symptoms of Parkinson's disease," *Movement Disorders*, vol. 26, pp. S42–S80, Oct. 2011.

[3] S. H. Fox, R. Katzenschlager, S.-Y. Lim, B. Ravina, K. Seppi, M. Coelho, W. Poewe, O. Rascol, C. G. Goetz, and C. Sampaio, "The Movement Disorder Society Evidence-Based Medicine Review Update: Treatments for the motor symptoms of Parkinson's disease," *Movement Disorders*, vol. 26, pp. S2–S41, Oct. 2011.

[4] S. L. Kowal, T. M. Dall, R. Chakrabarti, M. V. Storm, and A. Jain, "The current and projected economic burden of Parkinson's disease in the United States," *Movement Disorders*, vol. 28, pp. 311–318, Feb. 2013.

[5] J. M. Fisher, N. Y. Hammerla, T. Ploetz, P. Andras, L. Rochester, and R. W. Walker, "Unsupervised home monitoring of Parkinson's disease motor symptoms using body-worn accelerometers," *Parkinsonism & Related Disorders*, vol. 33, pp. 44–50, Dec. 2016.

[6] C. Godinho, J. Domingos, G. Cunha, A. T. Santos, R. M. Fernandes, D. Abreu, N. Gonçalves, H. Matthews, T. Isaacs, J. Duffen, A. Al-Jawad, F. Larsen, A. Serrano, P. Weber, A. Thoms, S. Sollinger, H. Graessner, W. Maetzler, and J. J. Ferreira, "A systematic review of the characteristics and validity of monitoring technologies to assess Parkinson's disease," *Journal of NeuroEngineering and Rehabilitation*, vol. 13, p. 24, Mar. 2016.

[7] T. O. Mera, D. A. Heldman, A. J. Espay, M. Payne, and J. P. Giuffrida, "Feasibility of home-based automated Parkinson's disease motor assessment," *Journal of neuroscience methods*, vol. 203, pp. 152–156, Jan. 2012.

[8] F. Parisi, G. Ferrari, M. Giuberti, L. Contin, V. Cimolin, C. Azzaro, G. Albani, and A. Mauro, "Inertial BSN-Based Characterization and Automatic UPDRS Evaluation of the Gait Task of Parkinsonians," *IEEE Transactions on Affective Computing*, vol. 7, pp. 258–271, July 2016.

[9] M. Giuberti, G. Ferrari, L. Contin, V. Cimolin, C. Azzaro, G. Albani, and A. Mauro, "Automatic UPDRS Evaluation in the Sit-to-Stand Task of Parkinsonians: Kinematic Analysis and Comparative Outlook on the Leg Agility Task," *IEEE journal of biomedical and health informatics*, vol. 19, pp. 803–814, May 2015.

[10] H. Jeon, W. Lee, H. Park, H. J. Lee, S. K. Kim, H. B. Kim, B. Jeon, and K. S. Park, "Automatic Classification of Tremor Severity in Parkinson's Disease Using a Wearable Device," *Sensors (Basel, Switzerland)*, vol. 17, Sept. 2017.

[11] Q. W. Oung, H. Muthusamy, H. L. Lee, S. N. Basah, S. Yaacob, M. Sarillee, and C. H. Lee, "Technologies for Assessment of Motor Disorders in Parkinson's Disease: A Review," *Sensors (Basel, Switzerland)*, vol. 15, pp. 21710–21745, Aug. 2015.

[12] W. Maetzler, J. Domingos, K. Srulijes, J. J. Ferreira, and B. R. Bloem, "Quantitative wearable sensors for objective assessment of Parkinson's disease," *Movement Disorders: Official Journal of the Movement Disorder Society*, vol. 28, pp. 1628–1637, Oct. 2013.

[13] G. Deuschl, J. Raethjen, R. Baron, M. Lindemann, H. Wilms, and P. Krack, "The pathophysiology of parkinsonian tremor: a review," *Journal of Neurology*, vol. 247 Suppl 5, pp. V33–48, Sept. 2000.

[14] R. Pearson, ronald.k.pearson@gmail.com, Y. Neuvo, J. Astola, and M. Gabbouj, "Generalized Hampel Filters," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, pp. 1–18, Aug. 2016.

[15] G. Rigas, D. Gatsios, D. I. Fotiadis, M. Chondrogiorgi, C. Tsironis, S. Konitsiotis, G. Gentile, A. Marcante, and A. Antonini, "Tremor UPDRS estimation in home environment," *Conference proceedings: ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual Conference*, vol. 2016, pp. 3642–3645, Aug. 2016.

[16] N. E. Piro, L. K. Piro, J. Kassubek, and R. A. Blechschmidt-Trapp, "Analysis and Visualization of 3d Motion Data for UPDRS Rating of Patients with Parkinsons Disease," *Sensors (Basel, Switzerland)*, vol. 16, June 2016.

[17] B. R. Chen, S. Patel, T. Buckley, R. Rednic, D. J. McClure, L. Shih, D. Tarsy, M. Welsh, and P. Bonato, "A Web-Based System for Home Monitoring of Patients With Parkinson #x0027;s Disease Using Wearable Sensors," *IEEE Transactions on Biomedical Engineering*, vol. 58, pp. 831–836, Mar. 2011.

Appendices

Table 1: Budget

	Quantity	Price Per Unit	Total
Patient Resources			
Patient Trial Processing	1	$100	$100
Computing Hardware			
Networking Device (Gateway)	2	$200	$400
Microcontroller	4	$100	$400
Electronics			
Breadboard	2	$10	$20
Battery (witch charging)	2	$30	$60
Circuit Elements Redundancy	1	$240	$240
Sensors			
Electrode pad (qty 300)	1	$70	$70
Accelerometer & Gyroscope	12	$10	$120
MyoWare Muscle Sensor (EMG)	3	$40	$120
Total			**$1,560**

Table 2: Machine Learning Algorithm Progress Table

Machine Learning Algorithm	Development	Pros	Cons
Linear Discriminant Analysis	Completed	75% accurate	Assumes normal distribution
Quadratic Discriminant Analysis	Completed	75% accurate	Less variance, but more bias
Linear Regression Closed Form	Completed	80% accurate	Easily overfit
Ridge Regression Closed Form	Completed	80% accurate	Prediction variance too large
Linear Regression Gradient Descent	Completed	83% accurate	Easily overfit
Ridge Regression Gradient Descent	Completed	84% accurate	Prediction variance too large
Logistic Regression	In progress	Increased accuracy	Long training times
Multiclass Logistic Regression	In progress	Increased accuracy	Long training times
Multiclass Neural Network	In progress	Increased accuracy	Long training times

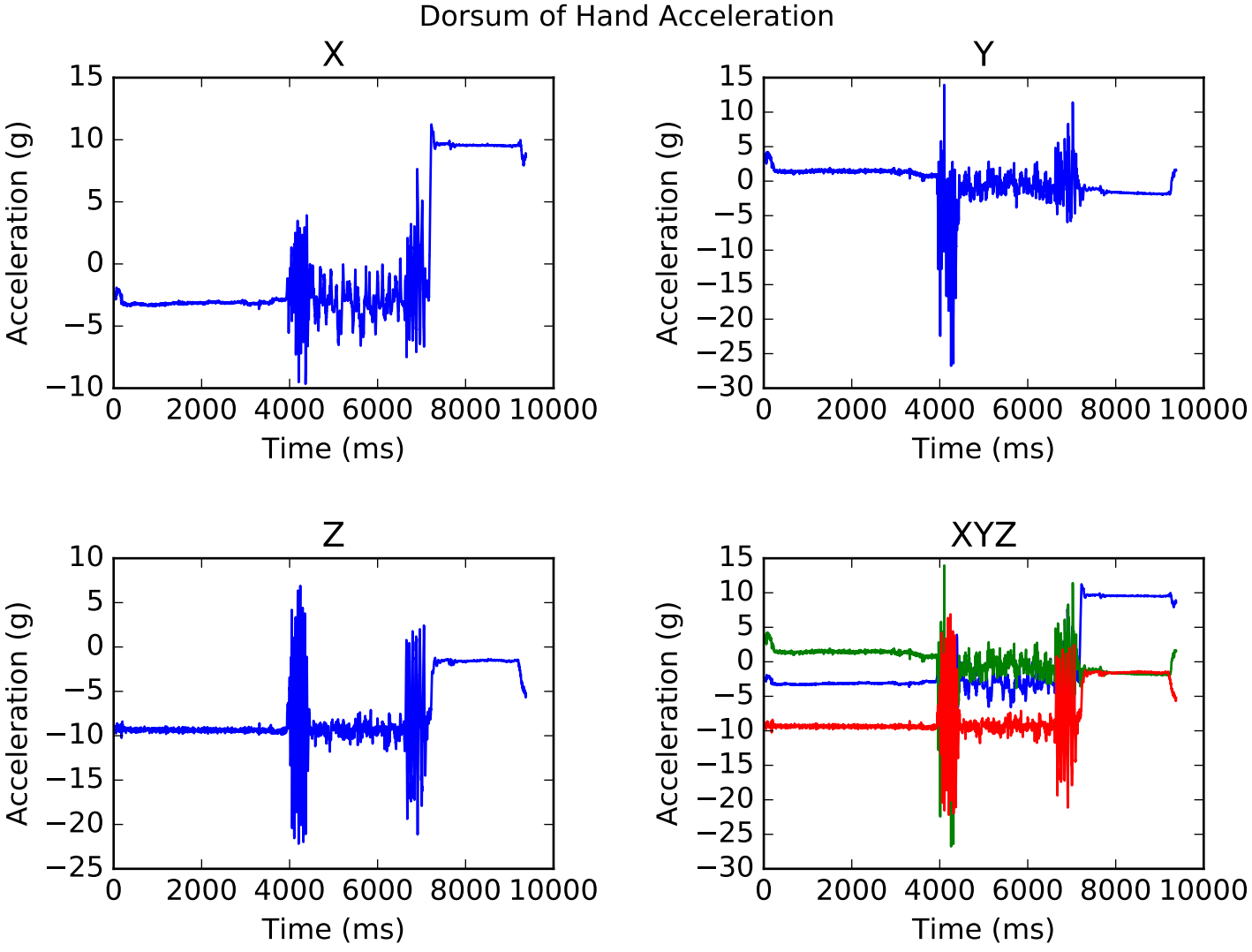
UPDA

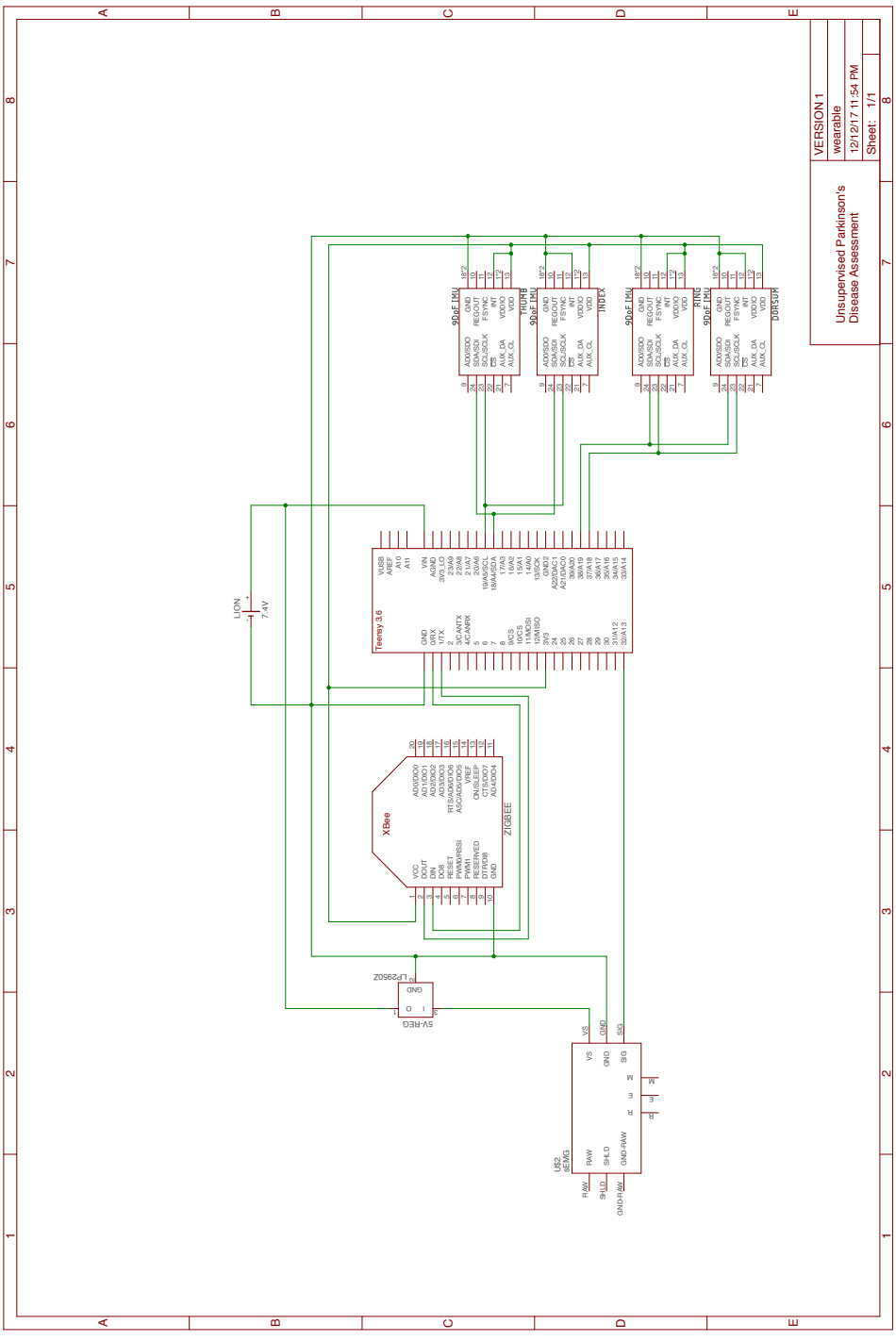
patient-1

Symptom Severity Report

2018-05-10 11:58:21.614969

Test	Percent of Runtime	Score
3.4. Finger Taps	53.23%	3.0
3.5. Hand Movements	30.45%	1.0
3.15. Postural Tremor	15.66%	0.0
3.16. Kinetic Tremor	13.99%	2.0
3.17. Rest Tremor Amplitude	72.22%	4.0
3.18. Consistency of Rest	43.45%	3.0


Dorsum of Hand Acceleration
X
Y
Z
XYZ
Acceleration (g)
Time (ms)
0
2000
4000
6000
8000
10000



12/12/17 11:57 PM f=0.70 /Users/Silversmith/Documents/eagle/upd/upd-wearable/wearable.sch (Sheet: 1/1)

Figure 1: Initial Prototype Wearable Circuit Design (V1)

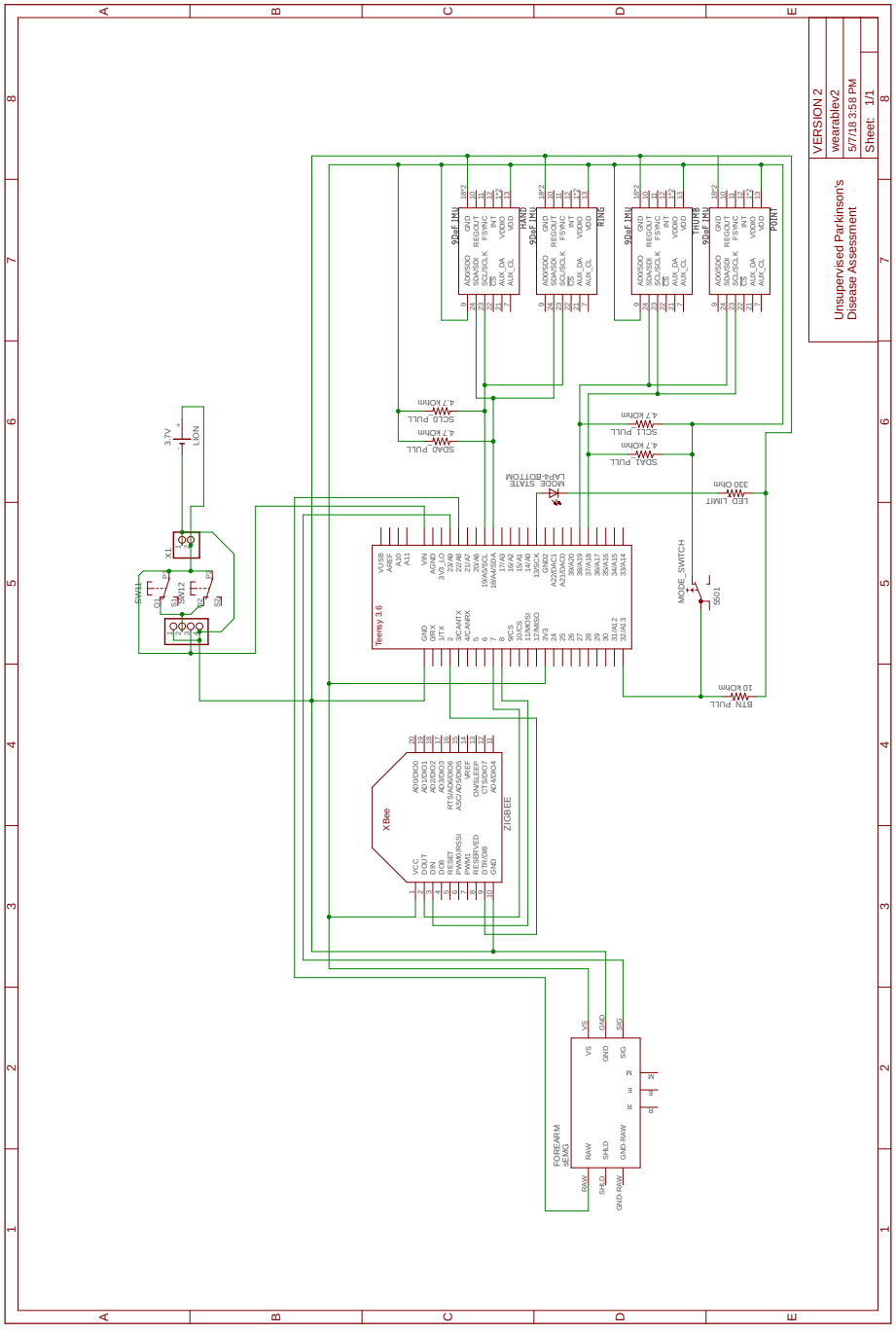


Figure 2: Final Prototype Wearable Circuit Design (V2)

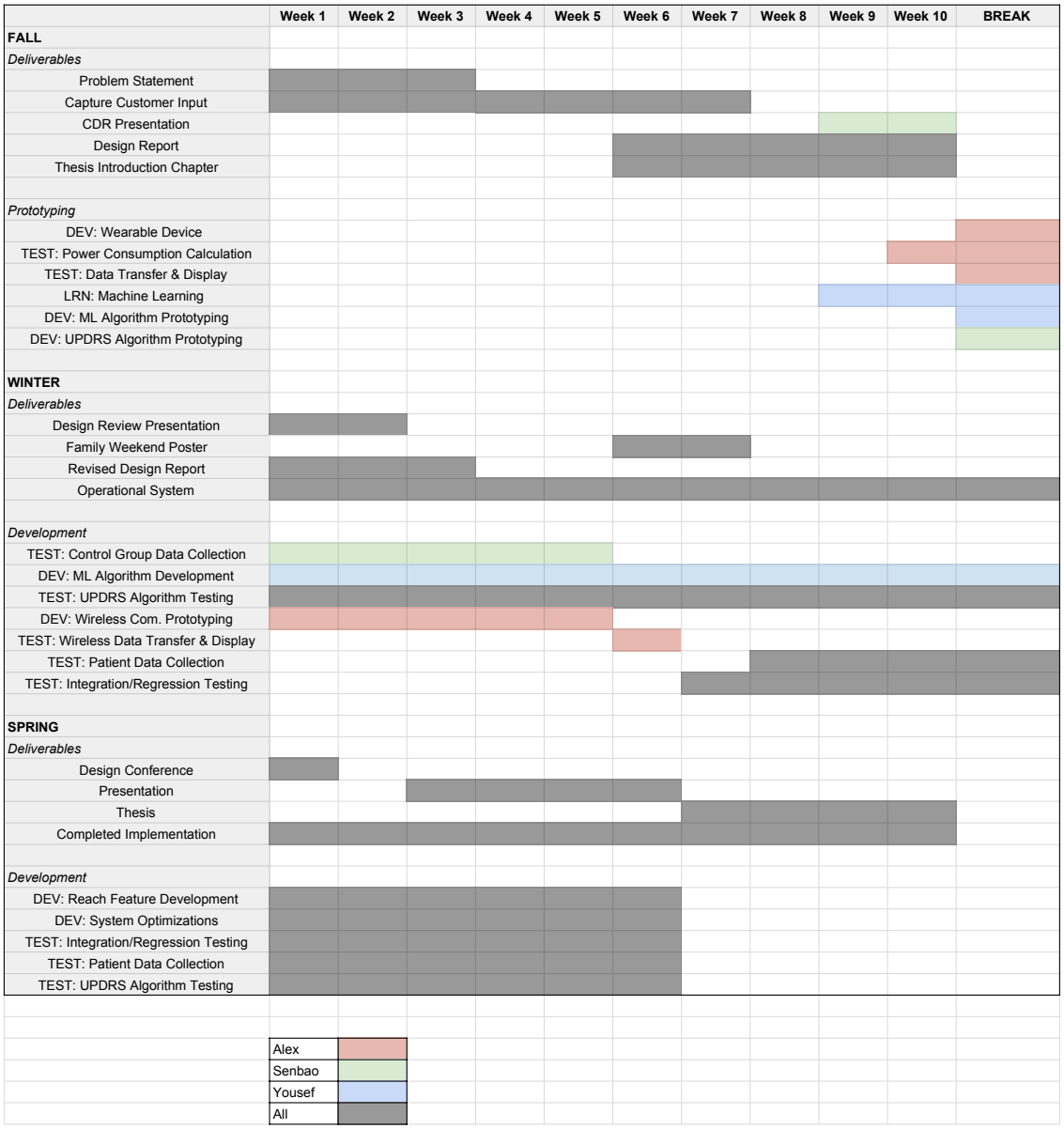


Figure 3: Timeline for Senior Design

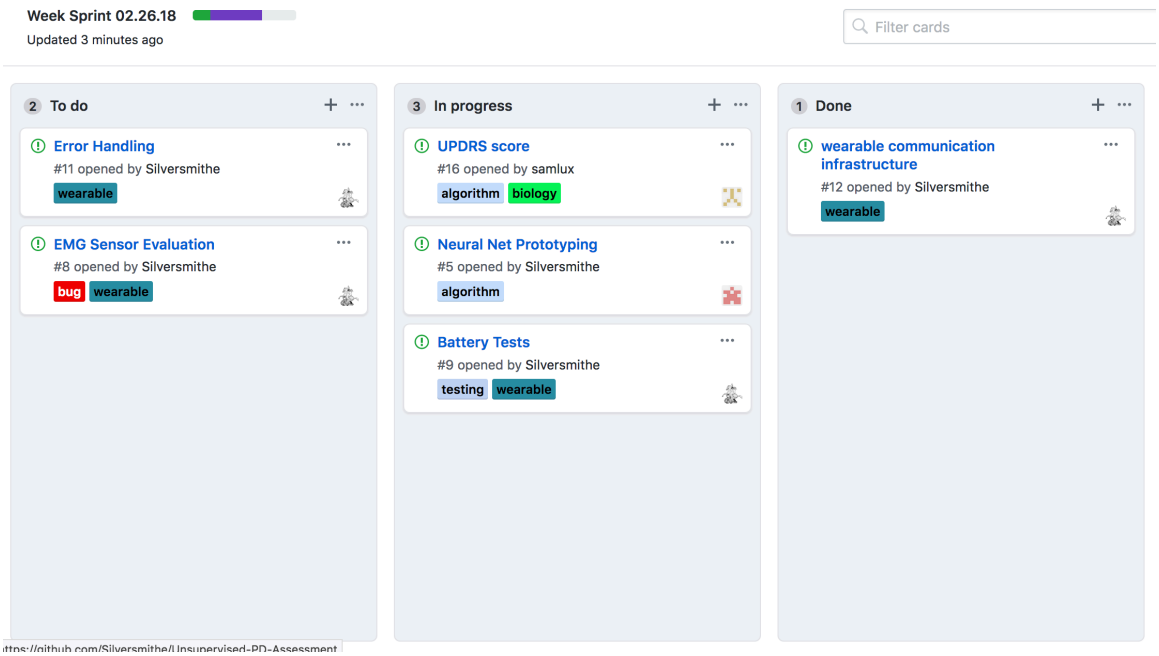


Figure 4: Team Kanban Board for Weekly Sprints

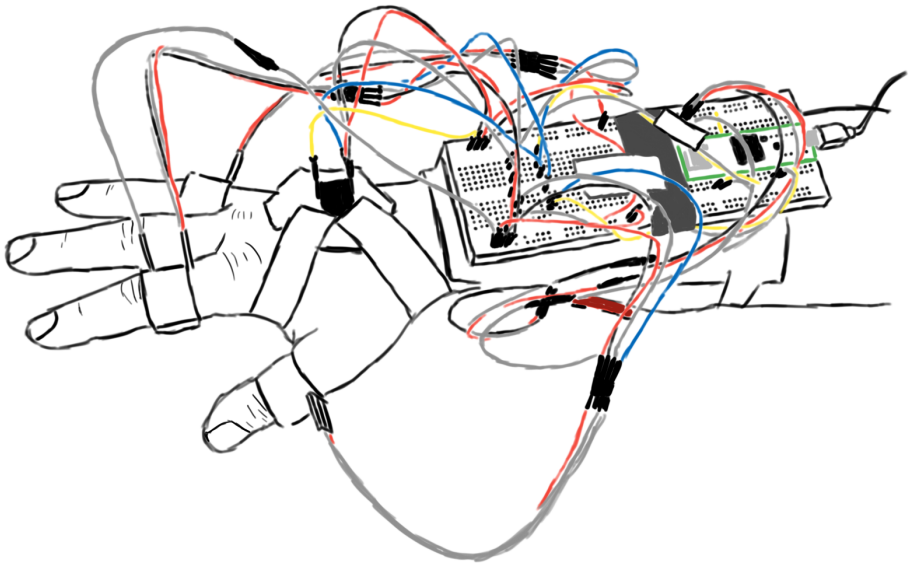


Figure 5: Sketch of Wearable Device Phase 1

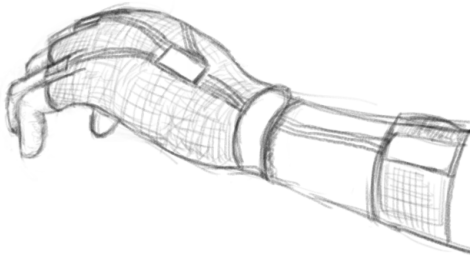


Figure 6: Concept Art of Wearable Device Phase 2

Wearable Code

```
/*-------------------------------------------------------------------------------
  file:         main.cpp (Wearable Version 2: SECOND SKIN)

  author:       Alexander Sami Adranly
  -------------------------------------------------------------------------------
  description:  Main Application for gathering and reporting information of both
  sensors in one. This is the prototype for the main application.
  Wearable device gathers information about the muscles of the arm and its
  fingers to perform diagnostics of parkinson's disease.

  In the H file, useful definitions will be made. The device can also be
  configured here to run or not run different sensors depending on what is
  necessary. This is particularly useful for unittesting the system.
  -------------------------------------------------------------------------------*/
#include "MyoEMG/MyoEMG.h"                // EMG library
#include "MPU9250/MPU9250.h"              // IMU library
#include "structures/IOBuffer.h"          // IOBuffer
#include "structures/Data.h"              // Data Struct
#include "com/com.h"                      // Communications Functions
#include "errors.h"                        // error values

#ifndef MAIN_H
#define MAIN_H

/* PROGRAM INFO */
#define VERSION 1
const uint8_t DEVICE_ID = 0x01;    // ID for this specific wearable device

/* DEVICE SELECTORS */
#define EMG_SELECT    true     // Turn on/off Forearm EMG readings
#define HAND_SELECT   true      // Turn on/off dorsum hand IMU readings
#define THUMB_SELECT  true      // Turn on/off Thumb IMU readings
#define POINT_SELECT  true      // Turn on/off Pointer IMU readings
#define RING_SELECT   true      // Turn on/off Ring IMU readings

/* COMMUNICATION DEFINITION */
const int SERVER_ADDR = 0xFE2F; // 0xFE2F
const int WEAR_ADDR = 0xFE31; // 0xFE31

#define BUFFER_SIZE    500
#define BUFFER_STALL   200
#define BUFFER_FLUSH   10       // how much the consumer is allowed to leave in
  buffer
#define CONSUMER_RATE  10       // miliseconds: amount of delay for consumer

/* IMU DEFINITION */
#define IMU_ADDR_LO 0x68        // Low address for IMU
#define IMU_ADDR_HI 0x69        // High address for IMU

/* SAMPLING INFORMATION */
#define DOUBLE_SAMPLE_RATE 5000     // microseconds, 200Hz
#define FULL_SAMPLE_RATE   10000    // microseconds, 100 Hz
#define DEMO_RATE          1000000  // microseconds
#define MODE_SW_TO         5000     // time to hold until switch
#define TRANSFER_POLL_TIME 5000    // time between each check

/* COMMUNICATION CONSTANTS */
const bool SERIAL_SELECT = false;       // Serial communication toggle
const bool XBEE_SELECT = false;         // Xbee (Radio) communication toggle

/* PINS */
const unsigned BUILTIN_LED = 13;    // builtin led for signaling
const unsigned LED_MODE_STAT = 21;  // led to display mode
const unsigned BTN_MODE = 32;       // button for switching modes
const unsigned EMG_RAW_PIN = 9;     // A9 analog pin for emg sampling
const unsigned EMG_RECT_PIN = 8;    // A8 analog pin for emg rectified sampling
const unsigned XBEE_SLEEP_PIN = 2;  // digital pin to sleep/wake radio
```

```
 67
 68 /* VARIABLES */
 69 const unsigned UC_SLEEP_TIME = 60000;  // 1 minute sleep cycle
 70
 71 /* FSM STATES */
 72 enum State {
 73   INIT,
 74   ONLINE,
 75   OFFLINE,
 76   KILL
 77 };
 78
 79 /* FUNCTION PROTOTYPES */
 80 void transfer_mode(void);
 81 void btn_isr(void);                        // read button presses
 82 bool imu_setup(bool trace);                // initialize all imus accordingly
 83 void sensor_isr(void);                     // called whenever the device samples
 84 void kill(void);                           // load the bootloader state
 85
 86 /* SLEEP FUNCTIONALITY */
 87 inline void sleepRadio(void);
 88 inline void wakeRadio(void);
 89 int sleepIMUs(void);
 90 int wakeIMUs(void);
 91 int powerNap(void);
 92
 93 #endif
 94
```

```
/*-------------------------------------------------------------------------------
  file:         errors.h

  author:       Alexander S. Adranly
  -------------------------------------------------------------------------------
  description:  Contains a list of different errors that could potentially show
                up during the runtime of the program. Based on these error
                messages, the device should be able to either recover or shut
                down accordingly
                This file will also be in control of error handling if such a
                thing is possible.

  errors:
                NONE                      No error exists

                IMU_ERROR                 Any issue reguarding the control of
                                          the IMU including communication,
                                          reading, data quality.

                EMG_ERROR                 issues reguarding the control and use
                                          of the EMG including communication,
                                          reading, and data quality

                ISOLATED_DEVICE_ERROR     issues reguarding the complete
                                          disconnection of the wearable from
                                          the server

                BUFFER_OVERFLOW           issues reguarding the use and
                                          functionality of the RAM buffer on
                                          the device.

                SD_ERROR                  Error recording information on the sd
                                          device

                FATAL_ERROR               Errors that cannot be recovered from
                                          Usually happens when there is another
                                          error in an uncompromising state
  -------------------------------------------------------------------------------*/
#ifndef ERRORS_H
#define ERRORS_H

enum ERROR {
  NONE,
  /* NEEDS TO KILL */
  FATAL_ERROR,
  /* SENSOR ERRORS */
  IMU_ERROR,
  EMG_ERROR,
  /* COMMUNICATION ERRORS */
  ISOLATED_DEVICE_ERROR,
  /* MEMORY ERRORS */
  BUFFER_OVERFLOW,
  /* SD CARD ERRORS */
  SD_ERROR,
};

#endif

```

```
/*-----------------------------------------------------------------------------
  file:         main.cpp (Wearable Version 2: Iron Fist)

  author:       Alexander S. Adranly
  -----------------------------------------------------------------------------
  description:  Main Application for gathering and reporting information of both
  sensors in one. This is the prototype for the main application.
  Wearable device gathers information about the muscles of the arm and its
  fingers to perform diagnostics of parkinson's disease.
  -----------------------------------------------------------------------------*/
#include "main.h"
#include "Arduino.h"              // Arduino Library
#include "stdint.h"               // Integer Library
#include "TimerOne.h"             // Timer Libaray
#include "Snooze/Snooze.h"        // uC sleep library

/* VARIABLES */
static IOBuffer BUFFER(BUFFER_SIZE);
static Data* temp_data;
static uint32_t __file_pos, __prev_pos;        // position of the data file
static bool __new_data;
SnoozeTimer alarmClock;                        // timer for sleeping
SnoozeBlock config(alarmClock);                // library to wake up

/* STATE */
volatile bool __sampling_mode;          // sampling (true), transferring (false)
volatile State __current_state;         // what peripherals can device use
volatile ERROR __error;                 // any complications
volatile bool __isr_buffer_overflow;    // isr-triggered error
volatile bool __enable_sampling;        // the button has been pushed
volatile bool __radio_sleeping;         // is the radio asleep?
volatile bool __imu_sleeping;           // are the imus sleeping

/* DEVICE INITIALIZATION */
bool __enabled[4] = {
  HAND_SELECT,
  RING_SELECT,
  POINT_SELECT,
  THUMB_SELECT
};

EMG forearm(EMG_RECT_PIN, EMG_RAW_PIN);
MPU9250 __imus[4] = {
  MPU9250(Wire, IMU_ADDR_HI),   // hand
  MPU9250(Wire, IMU_ADDR_LO),   // ring
  MPU9250(Wire1, IMU_ADDR_LO),   // pointer finger
  MPU9250(Wire1, IMU_ADDR_HI)   // thumb
};

/*
 * @function:     setup
 *
 * @description:  main initialization function, responsible creating all of the
 *                variables and doing the initial checks for hardware, networking,
 *                logging, and initializing the automata and error states.
 */
void setup(void) {
  bool hardware_success = true;
  bool network_success = false;
  __enable_sampling = false;
  __new_data = false; // usually false
  __file_pos = 0;
  __prev_pos = 0;
  __sampling_mode = false;
  __isr_buffer_overflow = false;
  __error = NONE;
  __current_state = INIT;
```

```
 68
 69   /* HARDWARE INITIALIZATION PROCEDURE */
 70   // 1. can you initialize all hardware?
 71   // STATE <- YES: INIT, NO: KILL
 72   pinMode(BUILTIN_LED, OUTPUT);
 73   pinMode(BTN_MODE, INPUT);
 74   pinMode(LED_MODE_STAT, OUTPUT);
 75   pinMode(XBEE_SLEEP_PIN, OUTPUT);
 76
 77   digitalWrite(XBEE_SLEEP_PIN, LOW);
 78   __radio_sleeping = false;
 79   __imu_sleeping = false;            // imus on
 80
 81   hardware_success = init_com(false) && hardware_success;
 82   hardware_success = imu_setup(false) && hardware_success;
 83   if(!hardware_success){
 84     __current_state = KILL;
 85     __error = IMU_ERROR;
 86     close_datastream();
 87     kill();
 88   }
 89
 90   /* NETWORK INITIALIZATION PROCEDURE */
 91   // 1. Can you contact the server?
 92   //    STATE <- YES: ONLINE, NO: OFFLINE
 93   if(XBEE_SELECT){
 94     log("checking network status...");
 95     wakeRadio();
 96     network_success = isAnyoneThere();
 97     sleepRadio();
 98   }
 99
100   /* decide if online or offline */
101   __current_state = (network_success)? ONLINE : OFFLINE;
102   if(__current_state == ONLINE){
103     log("state: online");
104     wakeRadio();
105   } else {
106     log("state: offline");
107   }
108
109   /* show device has booted successfully */
110   digitalWrite(LED_MODE_STAT, HIGH);
111
112   /* delay and signal before running */
113   for(int i=0; i<5; i++){
114     if(__current_state == ONLINE) { online_light(); }
115     else { offline_light(); }
116   }
117
118   /* declare start of device and current mode */
119   log("starting device...");
120   log("starting in transfer mode");
121
122   /* INITIALIZE TIMER FOR LOW POWER */
123   alarmClock.setTimer(UC_SLEEP_TIME);
124   /* INITIALIZE BUTTON INTERRUPT */
125   attachInterrupt(BTN_MODE, btn_isr, CHANGE);
126   /* INITIALIZE SENSOR INTERRUPT */
127   Timer1.initialize(FULL_SAMPLE_RATE);  // FULL_SAMPLE_RATE
128 }
129
130 /*
131  * @function:     loop
132  *
133  * @description:  main consumer thread, responsible for picking packets out
134  *                of the buffer and sending it over the radio or the serial
```

```
135  *                monitor.
136  */
137 void loop(void) {
138   /* error handling */
139   if(__error != NONE || __isr_buffer_overflow){
140     // isr generated interrupts
141     if(__isr_buffer_overflow){
142       __error = BUFFER_OVERFLOW;
143       __isr_buffer_overflow = false;
144     }
145 
146     switch (__error) {
147       case FATAL_ERROR:
148         close_datastream();
149         log("error: an fatal error has occurred...");
150         __current_state = KILL;
151         kill();
152         break;
153 
154       case ISOLATED_DEVICE_ERROR:
155         close_datastream();
156         log("error: device is unable to sustain a network connection...");
157         log("msg: transitioning to OFFLINE state");
158         __current_state = OFFLINE;
159         __error = NONE;
160         break;
161 
162       case BUFFER_OVERFLOW:
163         close_datastream();
164         log("error: I/O buffer has overflown...");
165         log("error: an fatal error has occurred...");
166         __current_state = KILL;
167         kill();
168         break;
169 
170       case SD_ERROR:
171         close_datastream();
172         if(SERIAL_SELECT){ Serial.println("error: a sd card error has occured...");
    }
173         __current_state = KILL;
174         kill_light();
175         noInterrupts();
176         while(1){ delay(10000); }
177         break;
178 
179       default: /* all other errors */
180         close_datastream();
181         log("error: an fatal error has occurred...");
182         __current_state = KILL;
183         kill();
184         break;
185     }
186   }
187 
188   /* enable interrupts */
189   if(__enable_sampling){
190     if(SERIAL_SELECT) {Serial.println("sampling enabled");}
191     Timer1.attachInterrupt(sensor_isr);
192     open_datastream();
193     __enable_sampling = false;
194     __new_data = true;
195   }
196 
197   /* mode behavior */
198   if(__sampling_mode){
199     /* allow for sampling BEHAVIOR */
200     if(!BUFFER.is_empty()){
```

```
201       noInterrupts();
202       temp_data = BUFFER.remove_front();
203       interrupts();
204
205       // write_console(temp_data);
206
207       /* DATA TRANSFER */
208       __error = log_payload(temp_data);
209
210     }
211   } else {
212     /* turn off sensor isr */
213     transfer_mode();
214   }
215 }
216
217 void transfer_mode(void){
218   /* take some time to sleep before doing anything */
219   powerNap();
220   delay(TRANSFER_POLL_TIME);
221
222   if(__current_state == OFFLINE && !__enable_sampling){
223     /* check for connection */
224     State temp = __current_state;
225     __current_state = (XBEE_SELECT && isAnyoneThere())? ONLINE: OFFLINE;
226     if(temp != __current_state){
227       if(__current_state == ONLINE){ log("state: online"); }
228       else { log("state: offline");}
229     }
230
231   } else if(__current_state == ONLINE && !__enable_sampling){
232     /* start sending data */
233     if(__new_data){
234       if (SERIAL_SELECT) {Serial.println("attempting data transfer...");}
235       __prev_pos = __file_pos;
236       __file_pos = write_to_server(__file_pos);
237       if(SERIAL_SELECT) {Serial.println(__file_pos);}
238
239       /* prevent device from trying to write */
240       if(__prev_pos == __file_pos){
241         __new_data = false;
242         if(SERIAL_SELECT) {Serial.println("no new data to transfer...");}
243       }
244     }
245   }
246
247 }
248
249 /*
250  * @function:     kill
251  *
252  * @description:  put the device in an infinite state of waiting and notify
253  *                the user that the device should be rebooted or debugged
254  */
255 void kill(void){
256   Timer1.detachInterrupt();
257   detachInterrupt(BTN_MODE);
258   close_datastream();
259   log("state: kill");
260   kill_light();
261   while(1){ delay(10000); }
262 }
263
264 /*
265  * @function:     imu_setup
266  *
267  * @param:        (bool) trace: turn on debugger tracer
```

```
268  *
269  * @description:  hardware initialization of the inertial measurement
270  *                units. should return some status of the operations.
271  *                Returns true if the initialization was 100% successful.
272  */
273 bool imu_setup(bool trace){
274   int status[4];
275   bool out = true;
276   if(trace && !SERIAL_SELECT) { return false; }
277
278   for(int i=0; i<4; i++){
279     if(__enabled[i]){
280       status[i] =  __imus[i].begin();
281       /* initializing components */
282       __imus[i].setAccelRange(MPU9250::ACCEL_RANGE_4G);
283
284       out = out && !(status[i] < 0);
285       if(trace && (status[i] < 0)){
286         log("imu hardware error!");
287         log("id: ");
288         log(String(i).c_str());
289         log("code: ");
290         log(String(status[i]).c_str());
291       }
292     }
293   }
294   return out;
295 }
296
297 /*
298  * @function:     sleepRadio
299  *
300  * @description:  Turn off the 802.15.4 radio
301  */
302 void sleepRadio(void){
303   if(!__radio_sleeping){
304     digitalWrite(XBEE_SLEEP_PIN, HIGH);
305     __radio_sleeping = true;
306   }
307 }
308
309 /*
310  * @function:     wakeRadio
311  *
312  * @description:  Turn on the 802.15.4 radio
313  */
314 void wakeRadio(void){
315   if(__radio_sleeping){
316     digitalWrite(XBEE_SLEEP_PIN, LOW);
317     __radio_sleeping = false;
318   }
319 }
320
321 int sleepIMUs(void){
322   if(!__imu_sleeping) {
323     for(unsigned i=0; i<4; i++)
324       if(__imus[i].sleep() < 0){ return -1 * i; }
325
326     __imu_sleeping = true;
327   }
328   return 0;
329 }
330
331 int wakeIMUs(void){
332   if(__imu_sleeping) {
333     for(unsigned i=0; i<4; i++)
334       if(__imus[i].wake() < 0){ return -1 * i; }
```

```
335
336     __imu_sleeping = false;
337   }
338   return 0;
339 }
340
341 int powerNap(void){
342   /* pass out for a little bit before working more */
343   Snooze.deepSleep(config);
344   return 0;
345 }
346
347 /*
348  * @function:     btn_isr
349  *
350  * @description:  function that is triggered when the button changes to HIGH
351  *                on the wearable device. this isr will wait for a period of
352  *                time, say 10 seconds, and if the button is still high after
353  *                that then the device will switch modes.
354  *                There are two modes to switch between:
355  *                sampling mode: collecting data
356  *                transfer mode: transferring data to the network if available
357  */
358 void btn_isr(void){
359   // pin should be high
360   unsigned value;
361   delay(250);
362   value = digitalRead(BTN_MODE);
363   if(value == 0){ return; } // exit if not 1
364
365   if(__sampling_mode){Timer1.detachInterrupt(); }
366
367   for(unsigned i=0; i < MODE_SW_TO; i+=1000){
368     digitalWrite(BUILTIN_LED, HIGH);
369     delay(500);
370     digitalWrite(BUILTIN_LED, LOW);
371     delay(500);
372   }
373   value = digitalRead(BTN_MODE);
374   delay(250);
375   /* constant val high -> change modes */
376   if(value == 1){
377     __sampling_mode = !__sampling_mode;
378     close_datastream(); // just in case a file is being written to
379     if(__sampling_mode){
380       log("switching to: sampling mode");
381       __enable_sampling = true;
382       // tell system there is more data
383       digitalWrite(LED_MODE_STAT, LOW);
384       sleepRadio();
385       wakeIMUs();
386
387     } else {
388       log("switching to: transfer mode");
389       digitalWrite(LED_MODE_STAT, HIGH);
390       wakeRadio();
391       sleepIMUs();
392     }
393   } else if(__sampling_mode){Timer1.attachInterrupt(sensor_isr); } // remove
    interrupt
394   delay(1000); // just in case
395 }
396
397 /*
398  * @function:     sensor_isr
399  *
400  * @description:  method that runs after each interrupt from the main thread.
```

```
401  *                this function is responsible for gathering all the information
402  *                from the sensors and store it in a packet, which gets pushed
403  *                onto the buffer.
404  */
405 void sensor_isr(void){
406   // new information set for buffer
407   Data packet = {
408     {0,0},                        // EMG DATA
409     {0,0,0,0,0,0,0,0,0},          // HAND
410     {0,0,0,0,0,0,0,0,0},          // THUMB
411     {0,0,0,0,0,0,0,0,0},          // POINT
412     {0,0,0,0,0,0,0,0,0}           // RING
413   };
414
415   if(EMG_SELECT){
416     packet.emg[0] = forearm.getRaw();
417     packet.emg[1] = forearm.getRect();
418   }
419
420   if(HAND_SELECT){
421     __imus[0].readSensor();
422     // accel
423     packet.hand[0] = __imus[0].getAccelX_mss();
424     packet.hand[1] = __imus[0].getAccelY_mss(); // getAccelX_mss
425     packet.hand[2] = __imus[0].getAccelZ_mss();
426     // gyro
427     packet.hand[3] = __imus[0].getGyroX_rads();
428     packet.hand[4] = __imus[0].getGyroY_rads(); // getGyroX_rads
429     packet.hand[5] = __imus[0].getGyroZ_rads();
430     // mag
431     packet.hand[6] = __imus[0].getMagX_uT();
432     packet.hand[7] = __imus[0].getMagY_uT(); // getMagX_uT
433     packet.hand[8] = __imus[0].getMagZ_uT();
434   }
435
436   if(THUMB_SELECT){
437     __imus[3].readSensor();
438     // accel
439     packet.thumb[0] = __imus[3].getAccelX_mss();
440     packet.thumb[1] = __imus[3].getAccelY_mss(); // getAccelX_mss
441     packet.thumb[2] = __imus[3].getAccelZ_mss();
442     // gyro
443     packet.thumb[3] = __imus[3].getGyroX_rads();
444     packet.thumb[4] = __imus[3].getGyroY_rads(); // getGyroX_rads
445     packet.thumb[5] = __imus[3].getGyroZ_rads();
446     // mag
447     packet.thumb[6] = __imus[3].getMagX_uT();
448     packet.thumb[7] = __imus[3].getMagY_uT(); // getMagX_uT
449     packet.thumb[8] = __imus[3].getMagZ_uT();
450   }
451
452   if(POINT_SELECT){
453     __imus[2].readSensor();
454     // accel
455     packet.point[0] = __imus[2].getAccelX_mss();
456     packet.point[1] = __imus[2].getAccelY_mss(); // getAccelX_mss
457     packet.point[2] = __imus[2].getAccelZ_mss();
458     // gyro
459     packet.point[3] = __imus[2].getGyroX_rads();
460     packet.point[4] = __imus[2].getGyroY_rads(); //getGyroX_rads
461     packet.point[5] = __imus[2].getGyroZ_rads();
462     // mag
463     packet.point[6] = __imus[2].getMagX_uT();
464     packet.point[7] = __imus[2].getMagY_uT(); // getMagX_uT
465     packet.point[8] = __imus[2].getMagZ_uT();
466   }
467
```

```
468   if(RING_SELECT){
469     __imus[1].readSensor();
470     // accel
471     packet.ring[0] = __imus[1].getAccelX_mss();
472     packet.ring[1] = __imus[1].getAccelY_mss(); // getAccelX_mss
473     packet.ring[2] = __imus[1].getAccelZ_mss();
474     // gyro
475     packet.ring[3] = __imus[1].getGyroX_rads();
476     packet.ring[4] = __imus[1].getGyroY_rads(); //getGyroX_rads
477     packet.ring[5] = __imus[1].getGyroZ_rads();
478     // mag
479     packet.ring[6] = __imus[1].getMagX_uT();
480     packet.ring[7] = __imus[1].getMagY_uT(); // getMagX_uT
481     packet.ring[8] = __imus[1].getMagZ_uT();
482   }
483
484   if(!BUFFER.push_back(packet)){ __isr_buffer_overflow = true; }
485 }
486
```

```
/*-----------------------------------------------------------------------------
  file:         Data.h

  author:       Alexander S. Adranly
  ----------------------------------------------------------------------------
  description:  This struct contains the all the information gathered and
                produced by the wearable device and other processing algorithms

                (16*2) + (4*9)*4 = 176
  ----------------------------------------------------------------------------*/
#include "stdint.h"

#ifndef DATA_H
#define DATA_H

struct Data {
  /* ------------------ EMG ------------------------*/
  int16_t emg[2]; // [Raw Rect]
  /* ------------------ IMU ------------------------
    (float[10])[Axyz Gxyz Mxyz T]
  */
  float hand[9];
  float thumb[9];
  float point[9];
  float ring[9];
};

typedef struct Data Data;

#endif

```

```
 1 /*-------------------------------------------------------------------------------
 2   file:         IOBuffer.h
 3
 4   author:       Alexander S. Adranly
 5   ------------------------------------------------------------------------------
 6   description:  An ADT that works as a circular data buffer for one consumer
 7                 and one producer.
 8   ------------------------------------------------------------------------------*/
 9 #include "stdint.h"
10 #include "Data.h"
11
12 #ifndef IOBUFFER_H
13 #define IOBUFFER_H
14
15 class IOBuffer {
16 public:
17   IOBuffer(unsigned bsize);
18   ~IOBuffer(void);
19
20   /* ACCESSORS */
21   inline unsigned num_elts(void){ return count; }
22   inline bool is_full(void){ return count == SIZE; }
23   inline bool is_empty(void){ return count == 0; }
24
25   /* MUTATORS */
26   Data* remove_front(void); // consuming an item from the front
27   bool push_back(Data item);     // producing an item and putting it in back
28
29 private:
30   Data** buffer;   // array for storing all the data
31   unsigned SIZE;              // Buffer's static size
32   unsigned pfront, pback;     // Pointers for the program
33   unsigned count;             // how many items are in the buffer currently
34 };
35
36 #endif
37
```

```
/*-------------------------------------------------------------------------------
  file:         IOBuffer.cpp

  author:       Alexander S. Adranly
  -------------------------------------------------------------------------------
  description:  An ADT that works as a circular data buffer for one consumer
                and one producer.
  -------------------------------------------------------------------------------*/
#include "IOBuffer.h"

/*
 * @function:       IOBuffer::IOBuffer
 *
 * @param:          (uint8_t) bsize: size of the buffer to be created
 *
 * @description:    * constructor *
 *                  create a circular data buffer of a specified
 *                  size and initialize all of its different components
 */
IOBuffer::IOBuffer(unsigned bsize){
  SIZE = bsize;
  pfront = pback = count = 0;

  buffer = new Data*[SIZE];
  for(unsigned i=0; i<SIZE; i++) { buffer[i] = new Data; }
}

/*
 * @function:          IOBuffer::~IOBuffer
 *
 * @description:       * destructor *
 *                     deallocate all of the allocated memory from the
 *                     constructur
 */
IOBuffer::~IOBuffer(void){
  for(unsigned i=0; i<SIZE; i++) { delete buffer[i]; }
  delete[] buffer;
}

/*
 * @function:             IOBuffer::remove_front
 *
 * @description:          enables the consumer to remove data points from the
 *                        circular buffer. changes the indicies so that the
 *                        frontmost datapoint can be overwritten as well as
 *                        returned to the calling program for use.
 *                        ASSUMING CHECKS ARE DONE FIRST
 *
 *  @return: (Data*):     a pointer to the frontmost data point in the circular
 *                        buffer
 */
Data* IOBuffer::remove_front(void){
  Data* temp = buffer[pfront];
  pfront = (pfront + 1) % SIZE;
  count--;
  return temp;
}

/*
 * @function:              IOBuffer::push_back
 *
 * @param:                 (Data) item: payload to push onto the buffer
 *
 * @description:           attempts to place a new data point into the circular
 *                         buffer.
 *
 * @return: (bool):        a boolean referring to the success or failure of the
```

```
 68  *                          calling program to insert a data point into the
 69  *                          circular buffer
 70  */
 71 bool IOBuffer::push_back(Data item){
 72   if(count >= SIZE){ return false; } // if full do not add
 73
 74   /* deep copy */
 75   for(int i=0; i<2; i++)          // emg
 76     buffer[pback]->emg[i] = item.emg[i];
 77
 78   for(int i=0; i<9; i++){        // imu
 79     buffer[pback]->hand[i] = item.hand[i];
 80     buffer[pback]->thumb[i] = item.thumb[i];
 81     buffer[pback]->point[i] = item.point[i];
 82     buffer[pback]->ring[i] = item.ring[i];
 83   }
 84
 85   pback = (pback + 1) % SIZE;
 86   count++;
 87   return true;
 88 }
 89
```

```
 1 /*-------------------------------------------------------------------------------
 2   file:         MyoEMG.h
 3
 4   author:       Alexander S. Adranly
 5   -------------------------------------------------------------------------------
 6   description:  A wrapper data structure to represent the EMG sensor attached
 7                 to the wearable device. responsible for keeping track of which
 8                 pins the device is attached to, which pin allows the device
 9                 to read raw data, and which pin allows the device to read
10                 rectified data.
11   -------------------------------------------------------------------------------*/
12 #include "Arduino.h"
13 #include "stdint.h"
14
15 #ifndef MYOEMG_H
16 #define MYOEMG_H
17
18 class EMG {
19 public:
20   EMG(uint8_t prect, uint8_t praw);             // EMG constructor
21   int16_t getRaw(void);                  // get the raw signal
22   int16_t getRect(void);                 // get the rectified signal
23
24 private:
25   uint8_t raw_pin;                              // define raw pin
26   uint8_t rect_pin;                             // define rectified pin
27 };
28
29 #endif
30
```

```
 1 /*-------------------------------------------------------------------------------
 2   file:         MyoEMG.cpp
 3
 4   author:       Alexander S. Adranly
 5   -------------------------------------------------------------------------------
 6   description:  A wrapper data structure to represent the EMG sensor attached
 7                 to the wearable device. responsible for keeping track of which
 8                 pins the device is attached to, which pin allows the device
 9                 to read raw data, and which pin allows the device to read
10                 rectified data.
11   -------------------------------------------------------------------------------*/
12 #include "Arduino.h"
13 #include "stdint.h"
14 #include "MyoEMG.h"
15
16 /*
17  * @function:        EMG::EMG
18  *
19  * @param:           (uint8_t) prect: analog GPIO pin that will read the rectified
   and
20  *                                    integrated signal
21  * @param:           (uint8_t) praw: analog GPIO pin that will read the raw emg
   signal
22  *
23  * @description:     constructor for the emg class, allows the user to specify
24  *                   both the raw pin and the rectified pin although both may not
25  *                   be used. However, it is mandatory for the user to specify a
26  *                   rectified pin at list. The constructor just stores the data
27  *                   for future collection
28  */
29 EMG::EMG(uint8_t prect, uint8_t praw):
30   raw_pin(praw),
31   rect_pin(prect)
32 {}
33
34 /*
35  * @function:      EMG::getRaw
36  *
37  * @description:   Function that polls the raw emg signal for the instantaneous value
38  *
39  * @return:        (int16_t) raw emg signal
40  */
41 int16_t EMG::getRaw(void){ return analogRead(raw_pin); }
42
43 /*
44  * @function:         EMG::getRect
45  *
46  * @description:      Function that polls rectified and integrated emg signal for the
47  *                    instantaneous value
48  *
49  * @return:           (int16_t) rectified and integrated emg signal
50  */
51 int16_t EMG::getRect(void){ return analogRead(rect_pin); }
52
```

File - /Users/Silversmith/PycharmProjects/Unsupervised-PD-Assessment/wearable/teensy/src/com/com.cpp

```
/*-------------------------------------------------------------------------------
  file:         com.cpp

  author:       Alexander S. Adranly
  -------------------------------------------------------------------------------
  description:  a library of functions that are designed for sending information
                and displaying information to the user. for instance, the device
                needs to be able to transmit data over Serial USB or Xbee to
                the server for future processing. Furthermore, the wearable
                device may want to react to an error by displaying an error
                message or flashing a light indicator.
  -------------------------------------------------------------------------------*/
#include "com.h"

/* xbee communication */
XBee xbee = XBee();

uint8_t broadcast[]     = { BROADCAST, DEVICE_ID};
uint8_t new_data[]      = { NEW_DATA, DEVICE_ID };
uint8_t continue_data[] = { CONTINUE_DATA, DEVICE_ID };
uint8_t char_buffer[PAYLOAD_SIZE];

Tx16Request tx_bc = Tx16Request(SERVER_ADDR, broadcast, sizeof(broadcast));
Tx16Request tx_nd = Tx16Request(SERVER_ADDR, new_data, sizeof(new_data));
Tx16Request tx_cd = Tx16Request(SERVER_ADDR, continue_data, sizeof(continue_data));
Tx16Request tx_char = Tx16Request(SERVER_ADDR, char_buffer, sizeof(char_buffer));

/* response information */
TxStatusResponse tx16 = TxStatusResponse();

uint8_t __missed_messages;
uint32_t __packet_counter; // FOR PACKET ACCOUNTABILITY EXPERIMENT

/* logger and SD card */
File __file;

/*
 * @function:            initialize all hardware connections for all mediums
 *                       of communication
 *
 * @description:         the function attempts to connect the wearable device
 *                       with the chosen method of communication and will react
 *                       accordingly if unable to do so. It will also always
 *                       open a connection with the SD card port as a place to
 *                       do logging
 */
bool init_com(bool erase){
  unsigned long start_time, current_time;
  bool hardware_success = true;

  /* SERIAL TESTING */
  if(SERIAL_SELECT){
    start_time = millis();
    Serial.begin(USB_BAUD);
    while(!Serial) {
      search_light();
      current_time = millis();
      if((current_time-start_time) > HW_TIMEOUT){
        hardware_success = false;
        break;
      }
    }
  }

  /* XBEE TESTING */
  if (XBEE_SELECT){
    start_time = millis();
```

```
 68     HWSERIAL.begin(RADIO_BAUD);
 69     while(!(HWSERIAL.availableForWrite() > 0)){
 70       search_light();
 71       current_time = millis();
 72       if((current_time-start_time) > HW_TIMEOUT){
 73         hardware_success = false;
 74         break;
 75       }
 76     }
 77     xbee.setSerial(HWSERIAL);
 78   }
 79
 80   /* SD TESTING */
 81   SD.begin(chip_select);
 82   /* initialize logger */
 83   __file = SD.open("log.txt", FILE_WRITE);
 84   if(__file){ /* file created successfully */
 85     __file.println("----- logfile -----");
 86     __file.close();
 87   } else { /* error creating file */
 88     hardware_success = false;
 89   }
 90
 91   /* clean slate with each new run */
 92   if(erase && SD.exists("data.txt")){ SD.remove("data.txt"); }
 93
 94   __file = SD.open("data.txt", FILE_WRITE);
 95   if(__file){ /* file created successfully */
 96     __file.close();
 97   } else { /* error creating file */
 98     hardware_success = false;
 99   }
100
101
102   /* initialize missed messages */
103   __missed_messages = 0;
104   __packet_id = 0;
105
106   return hardware_success;
107 }
108
109 /*
110  * @function:     isAnyoneThere
111  *
112  * @description:  have the radio try and contact the local server and wait for
113  *                a response. Return true or false depending on if you get a
114  *                response or not. (true) a server is there, (false) a server
115  *                is NOT there.
116  */
117 bool isAnyoneThere(void){
118   /* continue to send messages until you miss the limit */
119   __missed_messages = 0;
120   while(__missed_messages < MISSED_LIMIT){
121     xbee.send(tx_bc);
122     /* get the tx status response */
123     xbee.readPacket(TX_STAT_WAIT);
124     if(xbee.getResponse().getApiId() == TX_STATUS_RESPONSE){
125       xbee.getResponse().getTxStatusResponse(tx16);
126       if(tx16.getStatus() == SUCCESS){ return true; }
127     }
128     __missed_messages++;
129     delay(500);
130   }
131   return false;
132 }
133
134 /*
```

```
 * @function:     open_datastream
 *
 * @description:  Open the file for data logging so that it can be read
 *                constantly to flush out the buffer
 */
void open_datastream(void){
  if(!__file) {
    __file = SD.open("data.txt", FILE_WRITE);
    if(__file){ /* file created successfully */
      __file.println("----- datafile -----");
    }
  }
}

/*
 * @function:     close_datastream
 *
 * @description:  Close the file for data logging so that other files can be
 *                written to
 */
void close_datastream(void){
  if(__file) {
    __file.println();
    __file.close();
  }
}

/*
 * @function:     log
 *
 * @param:        (const char*) message:  message to send to log
 *
 * @description:  record the given message in a new line in the log
 */
void log(const char* message){
  if(SERIAL_SELECT) { Serial.println(message); }
  __file = SD.open("log.txt", FILE_WRITE);
  if(__file){
    __file.println(message);
    __file.close();
  }
}

/*
 * @function:     log_payload
 *
 * @param:        (Data*) src:  sample to record
 *
 * @description:  record the given message in a new line in the log
 */
ERROR log_payload(Data* src){
  // __file = SD.open("data.txt", FILE_WRITE);

  if(__file){
    // emg
    for(int iter=0; iter<2; iter++){
      __file.print(src->emg[iter]);
      if(iter < 1) { __file.print(" "); }
    }

    // hand
    __file.print(" ");
    for(int iter=0; iter<9; iter++){
      __file.print(src->hand[iter]);
      if(iter < 8) { __file.print(" "); }
    }

```

```
202     // thumb
203     __file.print(" ");
204     for(int iter=0; iter<9; iter++){
205       __file.print(src->thumb[iter]);
206       if(iter < 8) { __file.print(" "); }
207     }
208 
209     // point
210     __file.print(" ");
211     for(int iter=0; iter<9; iter++){
212       __file.print(src->point[iter]);
213       if(iter < 8) { __file.print(" "); }
214     }
215 
216     // ring
217     __file.print(" ");
218     for(int iter=0; iter<9; iter++){
219       __file.print(src->ring[iter]);
220       if(iter < 8) { __file.print(" "); }
221     }
222 
223     __file.println("");
224     // __file.close();
225   } else { return SD_ERROR; }
226 
227   return NONE;
228 }
229 
230 /*
231  * @function:    write_console
232  *
233  *  @param:       (Data*) src: a pointer to a data sample to print out to the
    console
234  *
235  *  @description: output the information of a given data point to the serial
236  *                console if the serial monitor has been activated
237  */
238 ERROR write_console(Data* src){
239   if(!SERIAL_SELECT){ return NONE; } // exit if the serial select has not been
    seleced
240 
241   // emg
242   Serial.print("(");
243   for(int iter=0; iter<2; iter++){
244     Serial.print(src->emg[iter]);
245     if(iter < 1) { Serial.print(", "); }
246   }
247   Serial.print(")\t");
248 
249   // hand
250   Serial.print("(");
251   for(int iter=0; iter<9; iter++){
252     Serial.print(src->hand[iter]);
253     if(iter < 8) { Serial.print(", "); }
254   }
255   Serial.print(")\t");
256 
257   // thumb
258   Serial.print("(");
259   for(int iter=0; iter<9; iter++){
260     Serial.print(src->thumb[iter]);
261     if(iter < 8) { Serial.print(", "); }
262   }
263   Serial.print(")\t");
264 
265   // point
266   Serial.print("(");
```

```
267   for(int iter=0; iter<9; iter++){
268     Serial.print(src->point[iter]);
269     if(iter < 8) { Serial.print(", "); }
270   }
271   Serial.print(")\t");
272 
273   // ring
274   Serial.print("(");
275   for(int iter=0; iter<9; iter++){
276     Serial.print(src->ring[iter]);
277     if(iter < 8) { Serial.print(", "); }
278   }
279   Serial.println(")");
280 
281   return NONE;
282 }
283 
284 bool match(unsigned& index, uint8_t* buffer, uint8_t val){
285   if(buffer[index] == val){
286     index++;
287     return true;
288   }
289   return false;
290 }
291 
292 bool header(unsigned& index, uint8_t* buffer){
293   bool result = true;
294   // '----- '
295   for(int i=0; i<5; i++)
296     result &= match(index, buffer, '-');
297 
298   result &= match(index, buffer, ' ');
299 
300   // match 'datafile'
301   result &= match(index, buffer, 'd');
302   result &= match(index, buffer, 'a');
303   result &= match(index, buffer, 't');
304   result &= match(index, buffer, 'a');
305   result &= match(index, buffer, 'f');
306   result &= match(index, buffer, 'i');
307   result &= match(index, buffer, 'l');
308   result &= match(index, buffer, 'e');
309 
310   // ' -----'
311   result &= match(index, buffer, ' ');
312   for(int i=0; i<5; i++)
313     result &= match(index, buffer, '-');
314 
315   return result;
316 }
317 
318 void whitepace(unsigned& index, uint8_t* buffer){
319   while(isWhitespace(buffer[index]) || isSpace(buffer[index]))
320       match(index, buffer, buffer[index]);
321 }
322 
323 uint16_t getDigit(uint8_t b){
324   switch (b) {
325     case '0': return 0;
326     case '1': return 1;
327     case '2': return 2;
328     case '3': return 3;
329     case '4': return 4;
330     case '5': return 5;
331     case '6': return 6;
332     case '7': return 7;
333     case '8': return 8;
```

```
334     case '9': return 9;
335     default: return 0;
336   }
337 }
338
339 uint16_t halfword(unsigned& index, unsigned size, uint8_t* buffer){
340   uint8_t start=index;
341   float e=-1.0;
342   uint16_t sum=0;
343
344   while(isDigit(buffer[index]) && index < size){
345     match(index, buffer, buffer[index]);
346     e=e+1.0;
347   }
348
349   for(unsigned i=start; i<index; i++){
350     sum += (getDigit(buffer[i]) * (uint16_t)(pow(10.0, e)));
351     e=e-1.0;
352   }
353   return sum;
354 }
355
356 uint32_t floating(unsigned& index, unsigned size, uint8_t* buffer){
357   unsigned start = index;
358   unsigned punct;
359   float pwr;
360   float sum=0.0;
361   bool negate = (match(index, buffer, '-'))? true: false;
362
363   if(isDigit(buffer[index])){
364     // digits here
365     while((isDigit(buffer[index]) || buffer[index] == '.') && index < size){
366       if(buffer[index] == '.'){ punct = index; } // save period location
367       match(index, buffer, buffer[index]);
368     }
369
370     pwr = ((float)punct - (float) start) - 1.0;
371     /* converting bytes to float */
372     for(unsigned i=start; i<punct; i++){
373       sum += ((float)getDigit(buffer[i]) * pow(10.0, pwr));
374       pwr = pwr - 1.0;
375     }
376
377     pwr = (float)punct - (float)(index-1);
378     for(unsigned i=index-1; i> punct; i--){
379       sum += ((float)getDigit(buffer[i]) * pow(10.0, pwr));
380       pwr = pwr + 1.0;
381     }
382
383   } else { return 0; } // error: cannot read digit
384
385   /* convert the float into a 32-bit value */
386   if(negate){ return (~((uint32_t)(sum*100.0))) + 1; }
387   return (uint32_t)(sum*100.0);
388 }
389
390
391 /*
392   read until a newline character
393
394   space should already be allocated (300 bytes)
395   does NOT store newline
396  */
397 unsigned read_line(uint8_t* buffer){
398   uint8_t lookahead;
399   unsigned count = 0;
400
```

```
401   /* clearing out the spaces */
402   if(__file.available()) {
403     lookahead = __file.read();
404     while(__file.available() && (lookahead == '\n' || lookahead == 13)){
405       lookahead = __file.read(); // skip to the next one
406     }
407   } else { return 0; } /* error */
408
409
410   /* actually going to read to file */
411   if(__file.available()){
412     while(__file.available()){
413       if(count >= FILE_BUFFER){ return count; } // too large!
414       if(count != 0) { lookahead = __file.read(); }
415       /* stop if nl | cr */
416       if(lookahead == '\n' || lookahead == 13){ break; }
417       buffer[count++] = lookahead;
418     }
419   } else { return 0; } /* error */
420
421   return count;
422 }
423
424 /*
425   parse the line
426
427   uint8_t: buffer of the line read
428
429   return boolean: is the line a header or not
430 */
431 bool parse_line(unsigned& size, uint8_t* buffer){
432   unsigned index = 0;
433   unsigned fill_ptr=0;
434   uint16_t emg[2];
435   uint32_t imu[36];
436
437   /* detect lookahead */
438   if(buffer[index] == '-' && header(index, buffer)){ return true; }
439
440   /* parse emg */
441   for(unsigned i=0; i<2; i++){
442     whitepace(index, buffer);
443     emg[i] = halfword(index, size, buffer);
444   }
445
446   /* parse imus */
447   for(unsigned i=0; i<36; i++){
448     whitepace(index, buffer);
449     imu[i] = floating(index, size, buffer);
450   }
451
452   /* load EMG into buffer */
453   for(unsigned i=0; i<2; i++){
454     buffer[fill_ptr++] = (uint8_t)((emg[i] >> 8) & 0x00FF);
455     buffer[fill_ptr++] = (uint8_t)((emg[i]) & 0x00FF);
456   }
457
458   /* load IMU into buffer */
459   for(unsigned i=0; i<36; i++){
460     buffer[fill_ptr++] = (uint8_t)((imu[i] >> 24) & 0x00FF);
461     buffer[fill_ptr++] = (uint8_t)((imu[i] >> 16) & 0x00FF);
462     buffer[fill_ptr++] = (uint8_t)((imu[i] >> 8) & 0x00FF);
463     buffer[fill_ptr++] = (uint8_t)((imu[i]) & 0x00FF);
464   }
465
466   return false;
467 }
```

```
468
469 bool write_line(unsigned size, uint8_t* buffer){
470   unsigned byte_size = 176;
471   unsigned offset = 2;
472   bool success = false;
473
474   char_buffer[0] = PAYLOAD;
475   char_buffer[1] = __packet_id;
476
477   /* first segment */
478   for(unsigned i=0; i<PAYLOAD_SIZE-offset; i++)
479     char_buffer[i+offset] = buffer[i];
480
481   /* send first packet */
482   while(__missed_messages < MISSED_LIMIT){
483     xbee.send(tx_char);
484     /* get the tx status response */
485     xbee.readPacket(TX_STAT_WAIT);
486     if(xbee.getResponse().getApiId() == TX_STATUS_RESPONSE){
487       xbee.getResponse().getTxStatusResponse(tx16);
488       if(tx16.getStatus() == SUCCESS){
489         success = true;
490         break;
491       }
492       delay(1000);
493     }
494     __missed_messages++;
495   }
496
497   /* wait before sending another message */
498   delay(500);
499
500   /* second segment */
501   for(unsigned i=0; i<byte_size-(PAYLOAD_SIZE-offset); i++)
502     char_buffer[i+offset] = buffer[i+(PAYLOAD_SIZE-offset)];
503
504   /* send second packet */
505   while(__missed_messages < MISSED_LIMIT){
506     xbee.send(tx_char);
507     /* get the tx status response */
508     xbee.readPacket(TX_STAT_WAIT);
509     if(xbee.getResponse().getApiId() == TX_STATUS_RESPONSE){
510       xbee.getResponse().getTxStatusResponse(tx16);
511       if(tx16.getStatus() == SUCCESS){
512         success = true;
513         break;
514       }
515       delay(1000);
516     }
517     __missed_messages++;
518   }
519   delay(500);
520
521   return success;
522 }
523
524 bool write_data_radio(bool isnew){
525   while(__missed_messages < MISSED_LIMIT){
526     if(isnew){ xbee.send(tx_nd); }
527     else { xbee.send(tx_cd); }
528
529     /* get the tx status response */
530     xbee.readPacket(TX_STAT_WAIT);
531     if(xbee.getResponse().getApiId() == TX_STATUS_RESPONSE){
532       xbee.getResponse().getTxStatusResponse(tx16);
533       if(tx16.getStatus() == SUCCESS){ return true; }
534       delay(1000);
```

```
535     }
536     __missed_messages = __missed_messages + 1;
537   }
538   return false;
539 }
540
541 /*
542  * @function:         write_to_server
543  *
544  * @description:      send all of the information for all the different
545  *                    sessions
546  *
547  *  note: return value is the position of the device in the file currently
548  *
549  */
550 uint32_t write_to_server(uint32_t position){
551   uint8_t* buffer = new uint8_t[FILE_BUFFER];
552   unsigned size;
553   uint32_t current_pos = position;
554   bool header = false;
555
556   if(__file){ __file.close(); }
557
558   if(SD.exists("data.txt")){
559
560     Serial.println();
561     Serial.println("ready to transmit...");
562     delay(5000);
563
564     /* should prepare file to be opened */
565     if(!__file){
566       __file = SD.open("data.txt");
567       if(__file.seek(current_pos)){
568         Serial.println("found the current position");
569       } else {
570         Serial.println("could not reach position!!");
571       }
572     }
573     digitalWrite(LED_MODE_STAT, LOW);
574
575     /*
576     PARSE A DATA SEGMENT
577
578     a segment consists of a header line
579     and multiple sample lines
580     */
581     if(__file){
582
583       /* check for the header */
584       __packet_id = 0;
585       if(__file.available()){
586         size = read_line(buffer);
587         header = parse_line(size, buffer);
588       }
589
590       /* header devices */
591       if(header){ write_data_radio(true); } /* send header signal */
592       else { write_data_radio(false); }
593
594       /* read samples */
595       while(__file.available()){
596         size = read_line(buffer);
597         header = parse_line(size, buffer);
598
599         if(header){ break; } // bread out of data segment
600
601         // WRITE TO THE "XBEE"
```

```
602         if(!write_line(size, buffer)) {
603           Serial.println("error writing to radio");
604           break;
605         }
606         __packet_id = (__packet_id + 1) % 200; // keep within the size of a file
607         delay(100); // slight delay is healthy for server (works well at 50)
608         // transfer_mode_light();
609         if(__packet_id == 0){ delay(5000); } // give the server breathing room
610       }
611
612       current_pos = __file.position();
613       __file.close();
614     }
615   } else {
616     log("no data exists to write");
617     if(SERIAL_SELECT){ Serial.println("no data exists to write"); }
618   }
619   // turn btn_mode light back on
620   digitalWrite(LED_MODE_STAT, HIGH);
621
622   delete[] buffer;
623   return current_pos;
624 }
625
626 /*
627  * @function:       online_light
628  *
629  * @description:    use the onboard light on the MCU to show the user that the
630  *                  device is running online (aka connected via xbee to the server)
631  */
632 void online_light(void){
633   digitalWrite(BUILTIN_LED, HIGH);
634   delay(500);
635   digitalWrite(BUILTIN_LED, LOW);
636   delay(500);
637 }
638
639 /*
640  * @function:       offline_light
641  *
642  * @description:    use the onboard light on the MCU to show the user that the
643  *                  device is running offline (aka NOT connected via xbee to the
    server)
644  */
645 void offline_light(void){
646   digitalWrite(BUILTIN_LED, HIGH);
647   delay(1000);
648   digitalWrite(BUILTIN_LED, LOW);
649   delay(1000);
650 }
651
652 /*
653  * @function:         search_light
654  *
655  * @param: (int) led: the digital io pin to control an LED
656  *
657  * @description:      command the specified LED to output this particular pattern
658  *                    to signal to the user that the device is searching for a
659  *                    communication medium to connect and use
660  */
661 void search_light(void){
662   digitalWrite(BUILTIN_LED, HIGH);
663   delay(100);
664   digitalWrite(BUILTIN_LED, LOW);
665   delay(100);
666   digitalWrite(BUILTIN_LED, HIGH);
667   delay(100);
```

```
668   digitalWrite(BUILTIN_LED, LOW);
669   delay(1000);
670 }
671
672 /*
673  * @function:         transfer_mode_light
674  *
675  * @param: (int) led: the digital io pin to control an LED
676  *
677  * @description:      light that specifies that a radio transfer is
678  *                    happening and that the power should NOT be turned off
679  */
680 void transfer_mode_light(void){
681   digitalWrite(BUILTIN_LED, HIGH);
682   delay(5);
683   digitalWrite(BUILTIN_LED, LOW);
684   delay(5);
685 }
686
687 /*
688  * @function:     kill_light
689  *
690  * @description:  a command to tell the builtin LED to light up and never turn
691  *                off to get the user's attention that the device needs to be
692  *                rebooted.
693  */
694 void kill_light(void){
695   digitalWrite(BUILTIN_LED, HIGH);
696 }
697
```

File - /Users/Silversmith/PycharmProjects/Unsupervised-PD-Assessment/wearable/teensy/src/com/com.h

```
/*-------------------------------------------------------------------------------
  file:         com.h

  author:       Alexander S. Adranly
  -------------------------------------------------------------------------------
  description:  a library of functions that are designed for sending information
                and displaying information to the user. for instance, the device
                needs to be able to transmit data over Serial USB or Xbee to
                the server for future processing. Furthermore, the wearable
                device may want to react to an error by displaying an error
                message or flashing a light indicator.
  -------------------------------------------------------------------------------*/
#include "Arduino.h"
#include "../structures/Data.h"
#include <XBee.h>
#include "../sd/SD.h"
#include <SPI.h>
#include <math.h>
#include "../errors.h"

#ifndef COM_H
#define COM_H

/* identification */
extern const uint8_t DEVICE_ID;

/* addressing */
extern const int SERVER_ADDR;
extern const int WEAR_ADDR;

/* hardware variables */
#define HW_TIMEOUT        5000          /* 10000ms : 10 seconds */
#define XBEE_INIT_TIMEOUT 500
#define XBEE_COM_TIMEOUT  10

/* serial information */
#define HWSERIAL     Serial3
#define USB_BAUD     115200
#define RADIO_BAUD   38400  // 57600
#define MISSED_LIMIT 100

/* buffer variables */
#define FILE_BUFFER  300

/* radio information */
#define PAYLOAD_SIZE 90  // max 100
#define TX_STAT_WAIT 1000

/* communication protocol variables */
/* should have a byte for identifying the message type */
// new_datafile
// continue_datafile
// sample/payload
// broadcast
const uint8_t BROADCAST     = 0x01;   // tell the server you are here
const uint8_t NEW_DATA      = 0x02;   // signify server to create new file
const uint8_t CONTINUE_DATA = 0x03;   // if not all data in segment transferred
const uint8_t PAYLOAD       = 0x04;
static uint8_t __packet_id;           // modulo 100

/* pins */
extern const bool SERIAL_SELECT;
extern const bool XBEE_SELECT;
extern const unsigned BUILTIN_LED;        /* builtin led on pin 13 */
extern const unsigned LED_MODE_STAT;      /* for controlling mode */

/* sd card communication */
```

Server Code

```
"""
[CLASS] Score

Date:       Tuesday June 12th, 2018
Author:     Senbao Lu and Yousef Zoumot


The Score object is responsible for using all of the
data generated from the previous filters and producing
an estimate of the patient's UPDRS score
"""
import numpy as np
from analysis.MahonyFilter import *
from MatrixBuilder import extract
import os


class Score(object):

    SAMPLING_PERIOD_1 = 33
    SAMPLING_PERIOD_2 = 50
    SAMPLING_PERIOD_3 = 100
    SAMPLING_PERIOD_4 = 300

    def __init__(self, filename):
        self.__filename = filename
        self.variable = None
        self.__num_instances = self.get_num_instances()

        self.__weights_ft_0hz = self.get_weights("W_ft_1_3hz.txt")
        self.__weights_ft_1hz = self.get_weights("W_ft_1hz.txt")
        self.__weights_ft_2hz = self.get_weights("W_ft_2hz.txt")
        self.__weights_ft_3hz = self.get_weights("W_ft_3hz.txt")

        self.__weights_ftin_0hz = self.get_weights("W_ftin_1_3hz.txt")
        self.__weights_ftin_1hz = self.get_weights("W_ftin_1hz.txt")
        self.__weights_ftin_2hz = self.get_weights("W_ftin_2hz.txt")
        self.__weights_ftin_3hz = self.get_weights("W_ftin_3hz.txt")

        self.__weights_hg_0hz = self.get_weights("W_hg_1_3hz.txt")
        self.__weights_hg_1hz = self.get_weights("W_hg_1hz.txt")
        self.__weights_hg_2hz = self.get_weights("W_hg_2hz.txt")
        self.__weights_hg_3hz = self.get_weights("W_hg_3hz.txt")

        self.__weights_hgin_0hz = self.get_weights("W_hgin_1_3hz.txt")
        self.__weights_hgin_1hz = self.get_weights("W_hgin_1hz.txt")
        self.__weights_hgin_2hz = self.get_weights("W_hgin_2hz.txt")
        self.__weights_hgin_3hz = self.get_weights("W_hgin_3hz.txt")

        self.result = {
            'name': str(filename).split(sep='/')[-1],
            'ftap':  [0.0, 0.0],
            'htap':  [0.0, 0.0],
            'ptrem': [0.0, 0.0],
            'ktrem': [0.0, 0.0],
            'rtrem': [0.0, 0.0],
            'crest': [0.0, 0.0]
        }

    def get_result(self):
        return self.result

    def process(self):
        """
        """
        weights1 = self.__weights_ft_0hz
        weights2 = self.__weights_ft_1hz
```

```
 68         weights3 = self.__weights_ft_2hz
 69         weights4 = self.__weights_ft_3hz
 70 
 71         inputs1 = self.get_input_1_3hz_test("yousef_5.txt")
 72         inputs2 = self.get_input_1hz_test("yousef_5.txt")
 73         inputs3 = self.get_input_2hz_test("yousef_5.txt")
 74         inputs4 = self.get_input_3hz_test("yousef_5.txt")
 75 
 76         dataset4 = self.get_predictions(inputs1, weights1)
 77         dataset3 = self.get_predictions(inputs2, weights2)
 78         dataset2 = self.get_predictions(inputs3, weights3)
 79         dataset1 = self.get_predictions(inputs4, weights4)
 80 
 81         weights5 = self.__weights_ftin_0hz
 82         weights6 = self.__weights_ftin_1hz
 83         weights7 = self.__weights_ftin_2hz
 84         weights8 = self.__weights_ftin_3hz
 85 
 86         inputs5 = self.get_input_1_3hz_test("yousef_6.txt")
 87         inputs6 = self.get_input_1hz_test("yousef_6.txt")
 88         inputs7 = self.get_input_2hz_test("yousef_6.txt")
 89         inputs8 = self.get_input_3hz_test("yousef_6.txt")
 90 
 91         dataset8 = self.get_predictions(inputs5, weights5)
 92         dataset7 = self.get_predictions(inputs6, weights6)
 93         dataset6 = self.get_predictions(inputs7, weights7)
 94         dataset5 = self.get_predictions(inputs8, weights8)
 95 
 96         weights9 = self.__weights_hg_0hz
 97         weights10 = self.__weights_hg_1hz
 98         weights11 = self.__weights_hg_2hz
 99         weights12 = self.__weights_hg_3hz
100 
101         inputs9 = self.get_input_1_3hz_test("yousef_7.txt")
102         inputs10 = self.get_input_1hz_test("yousef_7.txt")
103         inputs11 = self.get_input_2hz_test("yousef_7.txt")
104         inputs12 = self.get_input_3hz_test("yousef_7.txt")
105 
106         dataset12 = self.get_predictions(inputs9, weights9)
107         dataset11 = self.get_predictions(inputs10, weights10)
108         dataset10 = self.get_predictions(inputs11, weights11)
109         dataset9 = self.get_predictions(inputs12, weights12)
110 
111         weights13 = self.__weights_hgin_0hz
112         weights14 = self.__weights_hgin_1hz
113         weights15 = self.__weights_hgin_2hz
114         weights16 = self.__weights_hgin_3hz
115 
116         inputs13 = self.get_input_1_3hz_test("yousef_8.txt")
117         inputs14 = self.get_input_1hz_test("yousef_8.txt")
118         inputs15 = self.get_input_2hz_test("yousef_8.txt")
119         inputs16 = self.get_input_3hz_test("yousef_8.txt")
120 
121         dataset16 = self.get_predictions(inputs13, weights13)
122         dataset15 = self.get_predictions(inputs14, weights14)
123         dataset14 = self.get_predictions(inputs15, weights15)
124         dataset13 = self.get_predictions(inputs16, weights16)
125 
126         # print only htaps and ftaps for patients name
127         patient_name = self.__filename.split(sep="/")[-1]
128         if patient_name != 'data-3':
129             tapin_counter = self.count_tap_interuptions(dataset5, dataset6, dataset7
    , dataset8, self.__num_instances)
130             taps_counter = self.count_taps(dataset1, dataset2, dataset3, dataset4,
    self.__num_instances)
131 
132             graspin_counter = self.count_grasp_interuptions(dataset13, dataset14,
```

```
132 dataset15, dataset16, self.__num_instances)
133             grasp_counter = self.count_grasps(dataset9, dataset10, dataset11,
    dataset12, self.__num_instances)
134 
135             ratio1 = tapin_counter/taps_counter
136             ratio2 = graspin_counter/grasp_counter
137 
138             ##########################
139             # Finger Taps Scoring    #
140             ##########################
141             if ratio1 < 0.1:
142                 print("Finger Tap Score: 0")
143                 self.result['ftap'][1] = 0.0
144 
145             elif ratio1 <= 0.3:
146                 print("Finger Tap Score: 1")
147                 self.result['ftap'][1] = 1.0
148 
149             elif ratio1 <= 0.5:
150                 print("Finger Tap Score: 2")
151                 self.result['ftap'][1] = 2.0
152 
153             elif ratio1 <= 1:
154                 print("Finger Tap Score: 3")
155                 self.result['ftap'][1] = 3.0
156 
157             else:
158                 print("Finger Tap Score: 3")
159                 self.result['ftap'][1] = 3.0
160 
161             ##########################
162             # Hand Movements Scoring #
163             ##########################
164             if ratio2 <= 0.1:
165                 print("Hand Movement Score: 0")
166                 self.result['htap'][1] = 0.0
167 
168             elif ratio2 <= 0.3:
169                 print("Hand Movement Score: 1")
170                 self.result['htap'][1] = 1.0
171 
172             elif ratio2 <= 0.5:
173                 print("Hand Movement Score: 2")
174                 self.result['htap'][1] = 2.0
175 
176             elif ratio2 <= 1:
177                 print("Hand Movement Score: 3")
178                 self.result['htap'][1] = 3.0
179 
180             else:
181                 print("Hand Movement Score: 3")
182                 self.result['htap'][1] = 3.0
183 
184         if patient_name != 'data-1':
185 
186             self.score_time_tremor()
187 
188             # calculate tremor amplitude
189             self.calc_tremor_amplitude()
190 
191     def get_input(self):
192         """
193 
194         :return:
195         """
196         text_file = open("{}/raw.txt".format(self.__filename), "r")
197         lines = text_file.read().split("\n")
```

```
198         total_inputs = len(lines) - 1
199         text_file.close()
200 
201         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
202         output = np.zeros((total_inputs, 56))
203         for i in range(total_inputs):
204             dataset[i] = lines[i].split(' ')  # split data points of each instance
205 
206         # print(dataset_ftaps[1][0])
207         print(dataset[0])
208         print(dataset[0][54])
209         for i in range(total_inputs):
210             for k in range(55):
211                 if(k == 54):
212                     output[i][k] = 1
213                 else:
214                     try:
215                         output[i][k] = float(dataset[i][k])
216                     except ValueError:
217                         # print("Line {} is corrupt!".format(i))
218                         # print("column {} is corrupt!".format(k))
219                         break
220 
221         return output
222 
223     def get_input_test(self, textfile):
224         """
225 
226         :param textfile:
227         :return:
228         """
229         dir_path = os.getcwd()
230         text_file = open(str(dir_path) + "/resources/test_data/" + textfile , "r")
231         lines = text_file.read().split("\n")
232         total_inputs = len(lines) - 1
233         text_file.close()
234 
235         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
236         output = np.zeros((total_inputs, 56))
237         for i in range(total_inputs):
238             dataset[i] = lines[i].split(' ')  # split data points of each instance
239 
240         # print(dataset_ftaps[1][0])
241         print(dataset[0])
242         print(dataset[0][54])
243         for i in range(total_inputs):
244             for k in range(55):
245                 if(k == 54):
246                     output[i][k] = 1
247                 else:
248                     try:
249                         output[i][k] = float(dataset[i][k])
250                     except ValueError:
251                         # print("Line {} is corrupt!".format(i))
252                         # print("column {} is corrupt!".format(k))
253                         break
254 
255         return output
256 
257     def get_input_1hz(self):
258         """
259 
260         :return:
261         """
262         text_file = open("{}/raw.txt".format(self.__filename), "r")
263         lines = text_file.read().split("\n")
264         total_inputs = len(lines) - 1
```

```
265         text_file.close()
266 
267         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
268         output = np.zeros((total_inputs, 3801))
269         for i in range(total_inputs):
270             dataset[i] = lines[i]  # split data points of each instance
271 
272         temp_count = 0
273 
274         for i in range(int(total_inputs/100)):
275             for k in range(3801):
276                 if(k == 3800):
277                     output[i][k] = 1
278                 else:
279                     try:
280                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
281                     except ValueError:
282                         break
283 
284                 if k % 38 == 0:
285                     temp_count = temp_count + 1
286 
287         return output
288 
289     def get_input_1_3hz(self):
290         """
291 
292         :return:
293         """
294         text_file = open("{}/raw.txt".format(self.__filename), "r")
295         lines = text_file.read().split("\n")
296         total_inputs = len(lines) - 1
297         text_file.close()
298 
299         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
300         output = np.zeros((total_inputs, 11401))
301         for i in range(total_inputs):
302             dataset[i] = lines[i]  # split data points of each instance
303 
304         temp_count = 0
305 
306         for i in range(int(total_inputs/300)):
307             for k in range(11401):
308                 if k == 11400:
309                     output[i][k] = 1
310                 else:
311                     try:
312                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
313                     except ValueError:
314                         break
315 
316                 if k % 38 == 0:
317                     temp_count = temp_count + 1
318 
319         return output
320 
321     def get_input_2hz(self):
322         """
323 
324         :return:
325         """
326         text_file = open("{}/raw.txt".format(self.__filename), "r")
327         lines = text_file.read().split("\n")
328         total_inputs = len(lines) - 1
329         text_file.close()
```

```
330
331         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
332         output = np.zeros((total_inputs, 1901))
333         for i in range(total_inputs):
334             dataset[i] = lines[i]  # split data points of each instance
335
336         temp_count = 0
337
338         # output = float(dataset[1:2, 6:8, 15:17, 24:26, 33:35])
339
340         for i in range(int(total_inputs/50)):
341             for k in range(1901):
342                 if k == 1900:
343                     output[i][k] = 1
344                 else:
345                     try:
346                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
347                     except ValueError:
348                         # print("Line {} is corrupt!".format(i))
349                         # print("column {} is corrupt!".format(k))
350                         break
351                 if k % 38 == 0:
352                     temp_count = temp_count + 1
353
354         return output
355
356     def get_input_3hz(self):
357         """
358
359         :return:
360         """
361         text_file = open("{}/raw.txt".format(self.__filename), "r")
362         lines = text_file.read().split("\n")
363         total_inputs = len(lines) - 1
364         text_file.close()
365
366         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
367         output = np.zeros((total_inputs, 1255))
368         for i in range(total_inputs):
369             dataset[i] = lines[i]  # split data points of each instance
370
371         # print(dataset_ftaps[1][0])
372         # print(dataset[0])
373         # print(dataset[0][54])
374         temp_count = 0
375
376         # output = float(dataset[1:2, 6:8, 15:17, 24:26, 33:35])
377
378         for i in range(int(total_inputs/33)):
379             for k in range(1255):
380                 if(k == 1254):
381                     output[i][k] = 1
382                 else:
383                     try:
384                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
385                     except ValueError:
386                         # print("Line {} is corrupt!".format(i))
387                         # print("column {} is corrupt!".format(k))
388                         break
389                 if(k % 38 == 0 ):
390                     temp_count = temp_count + 1
391
392         return output
393
394     def get_input_1hz_test(self, textfile):
```

```
395         """
396 
397         :param textfile:
398         :return:
399         """
400         dir_path = os.getcwd()
401         text_file = open(str(dir_path) + "/resources/test_data/" + textfile , "r")
402         lines = text_file.read().split("\n")
403         total_inputs = len(lines) - 1
404         text_file.close()
405 
406         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
407         output = np.zeros((total_inputs, 3801))
408         for i in range(total_inputs):
409             dataset[i] = lines[i] # split data points of each instance
410         temp_count = 0
411 
412         for i in range(int(total_inputs/100)):
413             for k in range(3801):
414                 if k == 3800:
415                     output[i][k] = 1
416                 else:
417                     try:
418                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
419                     except ValueError:
420                         break
421                 if(k % 38 == 0 ):
422                     temp_count = temp_count + 1
423 
424         return output
425 
426     def get_input_1_3hz_test(self, textfile):
427         """
428 
429         :param textfile:
430         :return:
431         """
432         dir_path = os.getcwd()
433         text_file = open(str(dir_path) + "/resources/test_data/" + textfile , "r")
434         lines = text_file.read().split("\n")
435         total_inputs = len(lines) - 1
436         text_file.close()
437 
438         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
439         output = np.zeros((total_inputs, 11401))
440         for i in range(total_inputs):
441             dataset[i] = lines[i] # split data points of each instance
442 
443         # print(dataset_ftaps[1][0])
444         # print(dataset[0])
445         # print(dataset[0][54])
446         temp_count = 0
447 
448         # output = float(dataset[1:2, 6:8, 15:17, 24:26, 33:35])
449 
450         for i in range(int(total_inputs/300)):
451             for k in range(11401):
452                 if(k == 11400):
453                     output[i][k] = 1
454                 else:
455                     try:
456                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
457                     except ValueError:
458                         # print("Below values belong to 1_3hz")
459                         # print("Line {} is corrupt!".format(i))
```

```
460                         # print("column {} is corrupt!".format(k))
461                         break
462                 if(k % 38 == 0 ):
463                     temp_count = temp_count + 1
464 
465         return output
466 
467     def get_input_2hz_test(self, textfile):
468         """
469 
470         :param textfile:
471         :return:
472         """
473         dir_path = os.getcwd()
474         text_file = open(str(dir_path) + "/resources/test_data/" + textfile , "r")
475         lines = text_file.read().split("\n")
476         total_inputs = len(lines) - 1
477         text_file.close()
478 
479         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
480         output = np.zeros((total_inputs, 1901))
481         for i in range(total_inputs):
482             dataset[i] = lines[i]  # split data points of each instance
483         temp_count = 0
484 
485         for i in range(int(total_inputs/50)):
486             for k in range(1901):
487                 if(k == 1900):
488                     output[i][k] = 1
489                 else:
490                     try:
491                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
492                     except ValueError:
493                         break
494                 if k % 38 == 0:
495                     temp_count = temp_count + 1
496 
497         return output
498 
499     def get_input_3hz_test(self, textfile):
500         """
501 
502         :param textfile:
503         :return:
504         """
505         dir_path = os.getcwd()
506         text_file = open(str(dir_path) + "/resources/test_data/" + textfile, "r")
507         lines = text_file.read().split("\n")
508         total_inputs = len(lines) - 1
509         text_file.close()
510 
511         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
512         output = np.zeros((total_inputs, 1255))
513         for i in range(total_inputs):
514             dataset[i] = lines[i]  # split data points of each instance
515 
516         temp_count = 0
517 
518         for i in range(int(total_inputs/33)):
519             for k in range(1255):
520                 if k == 1254:
521                     output[i][k] = 1
522                 else:
523                     try:
524                         output[i][k] = float(dataset[temp_count % total_inputs][k %
    38])
```

```
525                     except ValueError:
526                         break
527                 if k % 38 == 0:
528                     temp_count = temp_count + 1
529 
530         return output
531 
532     def get_weights(self, textfile):
533         """
534 
535         :param textfile:
536         :return:
537         """
538         dir_path = os.getcwd()
539         text_file = open(str(dir_path) + "/resources/weights/" + textfile, "r")
540         lines = text_file.read().split("\n")
541         total_inputs = len(lines) - 1
542         text_file.close()
543 
544         # dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
545         dataset = np.zeros((total_inputs, 1))
546         # dataset = [[] for y in range(total_inputs)]
547         for i in range(total_inputs):
548             dataset[i][0] = float(lines[i])  # split data points of each instance
549         # print(dataset[0][0])
550 
551         # print(dataset_ftaps[1][0])
552         return dataset
553 
554     """
555     Sigmoid maps a number between 1 and 0
556     Sigmoid function:
557                     1
558     sigmoid(x)= ----------
559                 1 + e^(-x)
560     """
561     def sigmoid(self, temp_in):
562         return np.float64(1 / (1 + np.exp( - temp_in)))
563 
564     """
565     By multiplying the new inputs with our calculated
566     column weights, we generate a column of prediction
567     values to determine whether or not an action occured
568     """
569     def get_predictions(self, inputs, weights):
570         return self.sigmoid(np.matmul(inputs, weights))
571 
572     def get_num_instances(self):
573         """
574 
575         :return:
576         """
577         text_file = open("{}/raw.txt".format(self.__filename), "r")
578         lines = text_file.read().split("\n")
579         total_inputs = len(lines) - 1
580         text_file.close()
581 
582         return total_inputs
583 
584     def count_taps(self, dataset1, dataset2, dataset3, dataset4, total_instances):
585         """
586 
587         :param dataset1:
588         :param dataset2:
589         :param dataset3:
590         :param dataset4:
591         :param total_instances:
```

```
592         :return:
593         """
594         frequency_choice = 0
595         max_count = 0
596         count = 0
597 
598         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_1)):
599             if float(dataset1[i][0]) > 0.999:
600                 count = count + 1
601         if count > max_count:
602             frequency_choice = 1
603             max_count = count
604         count = 0
605 
606         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_2)):
607             if float(dataset2[i][0]) > 0.999:
608                 count = count + 1
609         if count > max_count:
610             frequency_choice = 2
611             max_count = count
612         count = 0
613 
614         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_3)):
615             if float(dataset3[i][0]) > 0.99:
616                 count = count + 1
617 
618         if count > max_count:
619             frequency_choice = 3
620             max_count = count
621         count = 0
622 
623         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_4)):
624 
625             if float(dataset4[i][0]) > 0.75:
626                 count = count + 1
627 
628         if count > max_count:
629             frequency_choice = 4
630             max_count = count
631 
632         count = 0
633 
634         if frequency_choice == 1:
635             print("The finger tap count is: ")
636             print(max_count)
637             print("\n")
638             print("using 3HZ frequency")
639             return max_count
640 
641         if frequency_choice == 2:
642             print("The finger tap count is: ")
643             print(max_count)
644             print("\n")
645             print("using 2HZ frequency")
646             return max_count
647 
648         if frequency_choice == 3:
649             print("The finger tap count is: ")
650             print(max_count)
651             print("\n")
652             print("using 1HZ frequency")
653             return max_count
654 
655         if frequency_choice == 4:
656             print("The finger tap count is: ")
657             print(max_count)
658             print("\n")
```

```
659             print("using 1/3HZ frequency")
660             return max_count
661 
662         if max_count == 0:
663             return print("no taps found, looking for tap interruptions")
664 
665     def count_tap_interuptions(self, dataset1, dataset2, dataset3, dataset4,
    total_instances):
666         """
667 
668         :param dataset1:
669         :param dataset2:
670         :param dataset3:
671         :param dataset4:
672         :param total_instances:
673         :return:
674         """
675         frequency_choice = 0
676         max_count = 0
677         count = 0
678 
679         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_1)):
680             if float(dataset1[i][0]) > 0.999:
681                 count = count + 1
682 
683         if count > max_count:
684             frequency_choice = 1
685             max_count = count
686         count = 0
687 
688         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_2)):
689             if float(dataset2[i][0]) > 0.999:
690                 count = count + 1
691 
692         if count > max_count:
693             frequency_choice = 2
694             max_count = count
695         count = 0
696 
697         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_3)):
698             if float(dataset3[i][0]) > 0.99:
699                 count = count + 1
700 
701         if count > max_count:
702             frequency_choice = 3
703             max_count = count
704         count = 0
705 
706         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_4)):
707 
708             if float(dataset4[i][0]) > 0.75:
709                 count = count + 1
710 
711         if count > max_count:
712             frequency_choice = 4
713             max_count = count
714         count = 0
715 
716         if frequency_choice == 1:
717             print("The finger tap interrupt count is: ")
718             print(max_count)
719             print("\n")
720             print("using 3HZ frequency")
721             return max_count
722 
723         if frequency_choice == 2:
724             print("The finger tap interrupt count is: ")
```

```
725             print(max_count)
726             print("\n")
727             print("using 2HZ frequency")
728             return max_count
729 
730         if frequency_choice == 3:
731             print("The finger tap interrupt count is: ")
732             print(max_count)
733             print("\n")
734             print("using 1HZ frequency")
735             return max_count
736 
737         if frequency_choice == 4:
738             print("The finger tap interrupt count is: ")
739             print(max_count)
740             print("\n")
741             print("using 1/3HZ frequency")
742             return max_count
743 
744         if max_count == 0:
745             print("no tap interruptions found, looking for grasps")
746 
747     def count_grasps(self, dataset1, dataset2, dataset3, dataset4, total_instances):
748         """
749 
750         :param dataset1:
751         :param dataset2:
752         :param dataset3:
753         :param dataset4:
754         :param total_instances:
755         :return:
756         """
757         frequency_choice = 0
758         max_count = 0
759         count = 0
760 
761         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_1)):
762 
763             if float(dataset1[i][0]) > 0.999:
764                 count = count + 1
765         if count > max_count:
766             frequency_choice = 1
767             max_count = count
768         count = 0
769 
770         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_2)):
771             if float(dataset2[i][0]) > 0.999:
772                 count = count + 1
773         if count > max_count:
774             frequency_choice = 2
775             max_count = count
776         count = 0
777 
778         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_3)):
779             if float(dataset3[i][0]) > 0.99:
780                 count = count + 1
781         if count > max_count:
782             frequency_choice = 3
783             max_count = count
784         count = 0
785 
786         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_4)):
787             if float(dataset4[i][0]) > 0.75:
788                 count = count + 1
789 
790         if count > max_count:
791             frequency_choice = 4
```

```
792             max_count = count
793         count = 0
794 
795         if frequency_choice == 1:
796             print("The hand grasp count is: ")
797             print(max_count)
798             print("\n")
799             print("using 3HZ frequency")
800             return max_count
801 
802         if frequency_choice == 2:
803             print("The hand grasp count is: ")
804             print(max_count)
805             print("\n")
806             print("using 2HZ frequency")
807             return max_count
808 
809         if frequency_choice == 3:
810             print("The hand grasp count is: ")
811             print(max_count)
812             print("\n")
813             print("using 1HZ frequency")
814             return max_count
815 
816         if frequency_choice == 4:
817             print("The hand grasp count is: ")
818             print(max_count)
819             print("\n")
820             print("using 1/3HZ frequency")
821             return max_count
822 
823         if max_count == 0:
824             return print("no hand grasps found, looking for grasp interruptions")
825 
826     def count_grasp_interuptions(self, dataset1, dataset2, dataset3, dataset4,
    total_instances):
827         """
828 
829         :param dataset1:
830         :param dataset2:
831         :param dataset3:
832         :param dataset4:
833         :param total_instances:
834         :return:
835         """
836         frequency_choice = 0
837         max_count = 0
838         count = 0
839 
840         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_1)):
841             if float(dataset1[i][0]) > 0.999:
842                 count = count + 1
843         if count > max_count == 1:
844             max_count = count
845         count = 0
846 
847         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_2)):
848             if float(dataset2[i][0]) > 0.999:
849                 count = count + 1
850         if count > max_count:
851             frequency_choice = 2
852             max_count = count
853         count = 0
854 
855         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_3)):
856             if float(dataset3[i][0]) > 0.99:
857                 count = count + 1
```

```
858         if count > max_count:
859             frequency_choice = 3
860             max_count = count
861         count = 0
862 
863         for i in range(0, int(total_instances/self.SAMPLING_PERIOD_4)):
864             if float(dataset4[i][0]) > 0.75:
865                 count = count + 1
866         if count > max_count:
867             frequency_choice = 4
868             max_count = count
869         count = 0
870 
871         if frequency_choice == 1:
872             print("The hand grasp interrupt count is: ")
873             print(max_count)
874             print("\n")
875             print("using 3HZ frequency")
876             return max_count
877 
878         if frequency_choice == 2:
879             print("The hand grasp interrupt count is: ")
880             print(max_count)
881             print("\n")
882             print("using 2HZ frequency")
883             return max_count
884 
885         if frequency_choice == 3:
886             print("The hand grasp interrupt count is: ")
887             print(max_count)
888             print("\n")
889             print("using 1HZ frequency")
890             return max_count
891 
892         if frequency_choice == 4:
893             print("The hand grasp interrupt count is: ")
894             print(max_count)
895             print("\n")
896             print("using 1/3HZ frequency")
897             return max_count
898 
899         if max_count == 0:
900             return print("no hand grasp interruptions found")
901 
902     def score_time_tremor(self):
903 
904         # higher sample number will give us higher accuracy later
905         # may need to change this later
906         text_file = open("{}/bandpass.txt".format(self.__filename), "r")
907         lines = text_file.read().split("\n")
908         total_inputs = len(lines) - 1
909         text_file.close()
910 
911         dataset = [[float(0) for x in range(1)] for y in range(total_inputs)]
912         output = np.zeros((total_inputs, 28))
913         for i in range(total_inputs):
914             dataset[i] = lines[i].split(' ')
915 
916         for rows in range(total_inputs):
917             for columns in range(28):
918                 output[rows][columns] = float(dataset[rows][columns])
919 
920         true_data = output  # MAKE SURE TO PULL FROM FILE
921         channel_num = true_data.shape[1]  # explicitly set to 28 by line: 914
922 
923         expected_sample_num = 10
924         # fs = 100
```

```
925
926         sample_size = int(self.__num_instances/expected_sample_num)
927
928         # df = fs/sample_size
929         sample_num = int(self.__num_instances/sample_size)
930
931         test_data = np.zeros((sample_size, 24))
932
933         # used to determine if the data has tremors
934         tremor_amp = 3
935         tremor_count = 0
936
937         # s = slice(3,4,5,6,7,8,12,13,14,15,16,17,21,22,23,24,25,26,30,31,32,33,34,
    35)
938         for i in range(0, sample_num):
939             # true data should then be raw data
940             s = slice(((i)*sample_size) , ((i+1)*sample_size) , 1)
941             test_data = true_data[s][0:]
942             # test_data_fft = sp.fft(test_data)
943             mag = np.abs(test_data)
944
945             # freq = 0 : df : fs - df
946             # s2 = slice(beginning_index, end_index, 1)
947             # mag_tremor = np.linalg.norm(true_data[s2][0:])
948             mag_tremor_max = np.max(mag, axis=0)  # max value of mag
949             for j in range(24):
950                 if mag_tremor_max[j] > tremor_amp:
951                     tremor_count = tremor_count+1
952
953             tremor_time = tremor_count/(sample_num * channel_num)
954
955         if tremor_time == 0:
956             print('Constancy of Tremor: 0 : Normal')
957             self.result['crest'][1] = 0.0
958
959         elif tremor_time <= 0.25:
960             print('Constancy of Tremor: 1 : Slight')
961             self.result['crest'][1] = 1.0
962
963         elif tremor_time <= 0.5:
964             print('Constancy of Tremor: 2 : Mild')
965             self.result['crest'][1] = 2.0
966
967         elif tremor_time <= 0.75:
968             print('Constancy of Tremor: 3 : Moderate')
969             self.result['crest'][1] = 3.0
970
971         elif tremor_time <= 1:
972             print('Constancy of Tremor: 4 : Severe')
973             self.result['crest'][1] = 4.0
974
975         else:
976             print('Error')
977
978     def calc_tremor_amplitude(self):
979         # testing time: (slice)
980         # 3*gravity, 1*hampel, 1*roll
981         mat = extract(self.__filename, "GH", "MEk", "OHr")
982         mat = np.matrix(mat).transpose()
983
984         fs = 100
985         data_for_Postural_Ax = mat[:, 0]  # 67 Hand_Ax
986         data_for_Postural_Ay = mat[:, 1]  # 68 Hand_Ay
987         data_for_Postural_Az = mat[:, 2]  # 69 Hand_Az
988
989         data_for_Postural_EMG_rect = mat[:, 3]  # 79 EMG_rect
990         data_length = self.__num_instances
```

```
991         data_for_Postural_Vx = np.zeros((data_length, 1))
992 
993         for i in range(1, data_length):
994             data_for_Postural_Vx[i] = data_for_Postural_Vx[i-1] +
    data_for_Postural_Ax[i-1] * (1/fs)
995             # print(data_for_Postural_Vx[i])
996 
997         data_for_Postural_Vy = np.zeros((data_length, 1))
998 
999         for i in range(1, data_length):
1000            data_for_Postural_Vy[i] =data_for_Postural_Vy[i-1] +
    data_for_Postural_Ay[i-1] * (1/fs)
1001
1002        data_for_Postural_Vz = np.zeros((data_length,1))
1003
1004        for i in range(1, data_length):
1005            data_for_Postural_Vz[i] =data_for_Postural_Vz[i-1] +
    data_for_Postural_Az[i-1] * (1/fs)
1006
1007        sample_period=100  # 1 second, can change later
1008        sample_num = int(data_length/sample_period)
1009        # times experiencing these tremors
1010        testing_time_post = np.zeros((data_length,1))
1011        testing_time_rest = np.zeros((data_length,1))
1012        testing_time_kine = np.zeros((data_length,1))
1013        sample_data_avg = np.zeros((1, 3)) # 1,4
1014
1015        for i in range(1, sample_num+1):
1016            s = slice((i-1)*sample_period, i*sample_period,1)
1017            # sample_data = [ data_for_Postural_Vx[s][0], data_for_Postural_Vy[s][0
    ], data_for_Postural_Vz[s][0], data_for_Postural_EMG_rect[s][0]]
1018            sample_data_avg = [np.abs(np.average(data_for_Postural_Vx[s])), np.abs(
    np.average(data_for_Postural_Vy[s])), np.abs(np.average(data_for_Postural_Vz[s]))]
    # was another one here
1019            sample_data_avg = np.matrix(sample_data_avg)
1020            # print(sample_data_avg)
1021
1022            EMG_change = np.max(data_for_Postural_EMG_rect[s]) - np.min(
    data_for_Postural_EMG_rect[s])
1023            # print(EMG_change)
1024            if sample_data_avg.item((0,0)) < 200 and sample_data_avg.item((0,1)) <
    50 and sample_data_avg.item((0,2)) < 400: # 0.05
1025                if EMG_change > 30: # 10
1026                    testing_time_post[s] = 1 # [s][0]
1027                else:
1028                    testing_time_rest[s] = 1 # [s][0]
1029
1030            # (0,0) - - (0,1)
1031            if sample_data_avg.item((0,0)) < 200 and sample_data_avg.item((0,1)) <
    50 and sample_data_avg.item((0,2)) > 400: # 0.2
1032                testing_time_kine[s] = 1 # [s][0]
1033
1034        # real_testing_time = np.dot(testing_time, true_time[1:sample_num*
    sample_period, 0])
1035        # 3 steps does here calling the above function in order to do something
1036        # real_testing_time_post = np.dot(testing_time_post, data[57]) #57
1037
1038        # calculating amplitude
1039        raw_data_tremor_amplitude_post = np.multiply(np.expand_dims(mat[:, 4], axis
    =0), np.transpose(testing_time_post))
1040        raw_data_tremor_amplitude_kine = np.multiply(np.expand_dims(mat[:, 4], axis
    =0), np.transpose(testing_time_kine))
1041        raw_data_tremor_amplitude_rest = np.multiply(np.expand_dims(mat[:, 4], axis
    =0), np.transpose(testing_time_rest))
1042
1043        r_hand = 10  # centimeters
1044        # postural amplitude
```

```
1045         amplitude = np.abs(r_hand * (np.tan(np.abs(np.min(
     raw_data_tremor_amplitude_post))) + np.tan(np.abs(np.max(
     raw_data_tremor_amplitude_post)))))
1046
1047         if amplitude <= 0.1:
1048             print("Postural Tremor Score: 0 : Normal")
1049             self.result['ptrem'][1] = 0.0
1050
1051         elif amplitude <= 1:
1052             print("Postural Tremor Score: 1 : Slight")
1053             self.result['ptrem'][1] = 1.0
1054
1055         elif amplitude <= 3:
1056             print("Postural Tremor Score: 2 : Mild")
1057             self.result['ptrem'][1] = 2.0
1058
1059         elif amplitude <=10:
1060             print("Postural Tremor Score: 3 : Moderate")
1061             self.result['ptrem'][1] = 3.0
1062
1063         elif amplitude > 10:
1064             print("Postural Tremor Score: 4 : Severe")
1065             self.result['ptrem'][1] = 4.0
1066
1067         else:
1068             print("Postural Tremor amplitude error")
1069             self.result['ptrem'][1] = 4.0
1070
1071         # kinetic amplitude
1072         amplitude = 100 * np.abs(r_hand * (np.tan(np.abs(np.min(
     raw_data_tremor_amplitude_kine))) + np.tan(np.abs(np.max(
     raw_data_tremor_amplitude_kine)))))
1073
1074         if amplitude <= 0.1:
1075             print("Kinetic Tremor Score: 0 : Normal")
1076             self.result['ktrem'][1] = 0.0
1077
1078         elif amplitude <= 1:
1079             print("Kineti Tcremor Score: 1 : Slight")
1080             self.result['ktrem'][1] = 1.0
1081
1082         elif amplitude <= 3:
1083             print("Kinetic Tremor Score: 2 : Mild")
1084             self.result['ktrem'][1] = 2.0
1085
1086         elif amplitude <=10:
1087             print("Kinetic Tremor Score: 3 : Moderate")
1088             self.result['ktrem'][1] = 3.0
1089
1090         elif amplitude > 10:
1091             print("Kinetic Tremor Score: 4 : Severe")
1092             self.result['ktrem'][1] = 4.0
1093
1094         else:
1095             print("Kinetic Tremor amplitude error")
1096             self.result['ktrem'][1] = 4.0
1097
1098         # resting tremor
1099         amplitude = np.abs(r_hand * (np.tan(np.abs(np.min(
     raw_data_tremor_amplitude_rest))) + np.tan(np.abs(np.max(
     raw_data_tremor_amplitude_rest)))))
1100
1101         if amplitude <= 0.1:
1102             print("Rest Tremor Score: 0 : Normal")
1103             self.result['rtrem'][1] = 0.0
1104
1105         elif amplitude <= 1:
```

```
1106            print("Rest Tremor Score: 1 : Slight")
1107            self.result['rtrem'][1] = 1.0
1108
1109        elif amplitude <= 3:
1110            print("Rest Tremor Score: 2 : Mild")
1111            self.result['rtrem'][1] = 2.0
1112
1113        elif amplitude <=10:
1114            print("Rest Tremor Score: 3 : Moderate")
1115            self.result['rtrem'][1] = 3.0
1116
1117        elif amplitude > 10:
1118            print("Rest Tremor Score: 4 : Severe")
1119            self.result['rtrem'][1] = 4.0
1120
1121        else:
1122            print("Rest Tremor amplitude error")
1123            self.result['rtrem'][1] = 4.0
1124
1125    def postural_tremor(self, data):
1126        """
1127
1128        :param data:
1129        :return:
1130        """
1131        fs = 100
1132        data_for_Postural_Ax = data[67]  # 67 Hand_Ax
1133        data_for_Postural_Ay = data[68]  # 68 Hand_Ay
1134        data_for_Postural_Az = data[69]  # 69 Hand_Az
1135
1136        data_for_Postural_EMG_rect = data[79]  # 79 EMG_rect
1137        data_length = len(data_for_Postural_Ax)
1138
1139        data_for_Postural_Vx = np.zeros((data_length,1))
1140
1141        for i in range(1, data_length):
1142            data_for_Postural_Vx[i][0] = data_for_Postural_Vx[i-1][0] +
     data_for_Postural_Ax[i][0] * (1/fs)
1143
1144        data_for_Postural_Vy = np.zeros((data_length, 1))
1145
1146        for i in range(1, data_length):
1147            data_for_Postural_Vy[i][0]=data_for_Postural_Vy[i-1][0] +
     data_for_Postural_Ay[i][0] * (1/fs)
1148
1149        data_for_Postural_Vz = np.zeros((data_length,1))
1150
1151        for i in range(1, data_length):
1152            data_for_Postural_Vz[i][0]=data_for_Postural_Vz[i-1][0] +
     data_for_Postural_Az[i][1] * (1/fs)
1153
1154        sample_period=100
1155        sample_num = np.floor(data_length/sample_period)
1156        # times experiencing these tremors
1157        testing_time_post = np.zeros((sample_num*sample_period,1))
1158        testing_time_rest = np.zeros((sample_num*sample_period,1))
1159        testing_time_kine = np.zeros((sample_num*sample_period,1))
1160        sample_data_avg = np.zeros((1, 4))
1161
1162        for i in range(0, sample_num):
1163            sample_data = [ data_for_Postural_Vx[(i-1)*sample_period:i*
     sample_period - 1, 0], data_for_Postural_Vy[(i-1)*sample_period:i*sample_period - 1
     , 0], data_for_Postural_Vz[(i-1)*sample_period:i*sample_period - 1 ,0],
     data_for_Postural_EMG_rect[(i-1)*sample_period:i*sample_period-1, 0]]
1164            sample_data_avg[0] = np.avg(sample_data)
1165            EMG_change = np.max(sample_data[3]) - np.min(sample_data[3])
1166            if sample_data_avg[0] < 0.05 and sample_data_avg[2] < 0.05 and
```

```
1166 sample_data_avg[1] < 0.05:
1167                 if EMG_change > 10:
1168                     testing_time_post[(i-1)*sample_period:i*sample_period, 0] = 1
1169                 else:
1170                     testing_time_rest[(i-1)*sample_period:i*sample_period, 0] = 1
1171             if sample_data_avg[0] < 0.05 and sample_data_avg[2] < 0.05 and
     sample_data_avg[1] > 0.2:
1172                 testing_time_kine[(i-1)*sample_period:i*sample_period, 0] = 1
1173
1174         # real_testing_time = np.dot(testing_time, true_time[1:sample_num*
     sample_period, 0])
1175         #3 steps does here calling the above function in order to do something
1176         real_testing_time_post = np.dot(testing_time_post, data[57]) #57
1177         real_testing_time_kine = np.dot(testing_time_kine, data[57]) #57
1178         real_testing_time_rest = np.dot(testing_time_rest, data[57]) #57
1179
```

```
#! /usr/bin/python

"""
[SERVER] Unsupervised Parkinson's Disease Assessment

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

Server Related Classes
"""
# !/bin/bash
import os
import sys
import time
import serial
from threading import Thread, Lock
from MatrixBuilder import extract
from analysis.MahonyFilter import MahoneyFilter
from analysis.RawDataFilter import RawDataFilter
from PipelineManager import PipelineManager
from digi.xbee.devices import Raw802Device, RemoteRaw802Device
from digi.xbee.models.address import XBee16BitAddress

# GLOBALS
VERSION = 1
RUNNING = True
# SERVER
PORT = "/dev/ttyUSB0"
BAUD_RATE = 38400  # 57600
SERVER_16B_ADDR = "FE2F"
WEAR_16B_ADDR = "FE31"
# SD CARD
# LINUX
if sys.platform == 'darwin':
    SD_PATH = "/Volumes/UPDA-SD"
    SD_DATA_PATH = "/Volumes/UPDA-SD/DATA.txt"

else:
    SD_PATH = "/media/iron-fist/UPDA-SD"
    SD_DATA_PATH = "/media/iron-fist/UPDA-SD/DATA.txt"

"""
[CLASS] WEARABLE DEVICE

Keeps track of general statistics and characteristics of the radio
mounted on the wearable device in communication with this specific
server.
"""


class Wearable(object):

    def __init__(self):
        self.id = "unknown"
        self.address = XBee16BitAddress.from_hex_string("0x0000")
        self.received_count = 0
        self.sent_count = 0
        self.remote = None
        self.runtime = 0

    def reset(self):
        self.id = "unknown"
        self.address = XBee16BitAddress.from_hex_string("0x0000")
        self.received_count = 0
        self.sent_count = 0
        self.remote = None
        self.runtime = 0
```

```
 68
 69
 70 """
 71 GLOBAL VARIABLES
 72
 73 Used for IPC between threads as well as general book keeping for
 74 the entire console system with reference to the wearable device
 75 in communication.
 76 """
 77
 78 UpdaWear = Wearable()
 79 MessageBuffer = list()
 80 BufferLock = Lock()
 81
 82 """
 83 [CLASS] INSTANCE LOADER
 84
 85 takes all the raw data over the radio and passes it through a filter
 86 Listens to an array that is populated by the XBee server and uses it to
 87 filter and process instances for certain files
 88 """
 89
 90
 91 class InstanceLoader(Thread):
 92
 93     def __init__(self, file_count):
 94         Thread.__init__(self)
 95         self.__raw_instances = list()
 96         self.__file_count = file_count  # file count
 97         self.raw_filter = RawDataFilter()
 98         self.__file = None
 99
100     def run(self):
101         """
102         Given a list of samples, open up or append a new file and start
103         writing the information there
104         :return:
105         """
106         __running = True
107         __packet_extend = False
108         try:
109             print("starting instance loader thread...")
110             while __running:
111
112                 # if MessageBuffer past Sample threshold, take them and start
    processing them
113                 # eventually would be nice: len(MessageBuffer) >= 200
114                 if len(MessageBuffer) > 0 and len(self.__raw_instances) <= 1:
115                     with BufferLock:
116                         self.__raw_instances.extend(MessageBuffer)  # extend, do not
     overwrite what you already have
117                         MessageBuffer.clear()
118
119                 elif len(self.__raw_instances) == 1:
120                     instance = int(str(self.__raw_instances[0][0]), 16)
121                     if instance == self.raw_filter.OLD_DATASEG_MSG:
122                         print("msg: continuing previous session")
123                         self.__raw_instances.pop(0)
124
125                     if instance == self.raw_filter.CLOSE_MSG:
126                         __running = False
127                         self.__raw_instances.pop(0)
128                         continue
129
130                 elif len(self.__raw_instances) > 1:
131                     # process the values that are in your buffer
132                     # instances are byte arrays
```

```
133                     # payloads are twins
134
135                     while len(self.__raw_instances) > 1:
136                         # look at opcode
137                         instance = int(str(self.__raw_instances[0][0]), 16)
138
139                         ######################
140                         # BROADCAST MESSAGES #
141                         ######################
142                         if instance == self.raw_filter.BROADCAST_MSG:
143                             # okay cool, do not need to store
144                             print("msg: received device broadcast")
145                             pass
146
147                         ######################
148                         # OLD DATA MESSAGES #
149                         ######################
150                         elif instance == self.raw_filter.OLD_DATASEG_MSG:
151                             # okay cool, do not need to store FOR NOW
152                             print("msg: continuing previous session")
153                             pass
154
155                         ######################
156                         # NEW DATA MESSAGES #
157                         ######################
158                         elif instance == self.raw_filter.NEW_DATASEG_MSG:
159                             # okay need to open a new file
160                             self.__file_count += 1          # increment file count
    for new file name
161                             if self.__file is not None:     # close any previously
    open file
162                                 self.__file.close()
163
164                             # create a new folder for raw file
165                             if not os.path.exists("./data/data-{}".format(self.
    __file_count)):
166                                 os.mkdir("./data/data-{}".format(self.__file_count))
167
168                             # create a new file to write to
169                             self.__file = open("./data/data-{}/raw.txt".format(self.
    __file_count), "w")
170                             print("msg: created new data session")
171
172                         ####################
173                         # PAYLOAD MESSAGES #
174                         ####################
175                         elif instance == self.raw_filter.PAYLOAD_MSG:
176                             # pass data through payload filter
177                             if len(self.__raw_instances) > 1:
178                                 # check packet id's
179                                 if int(str(self.__raw_instances[0][1]), 16) == int(
    str(self.__raw_instances[1][1]), 16):
180                                     _, _, _data = self.raw_filter.process(self.
    __raw_instances[0], self.__raw_instances[1])
181                                     self.__raw_instances.pop(0)  # pop the first val
    , the next is popped at end of loop
182                                     # write to file
183                                     for value in _data:
184                                         self.__file.write('{}\t'.format(value))
185                                     self.__file.write('\n')  # end with a newline
186
187                                 else:
188                                     # second part of packet did not show up, so the
    packet is not useful
189                                     print("warning: unable to find packet pair")
190                                     print("{} {}".format(int(str(self.
    __raw_instances[0][1]), 16), int(str(self.__raw_instances[1][1]),16)))
```

```
191                                     # continue  # see if it shows up later
192
193                             else:
194                                 # if there is 1 or zero packets, do not throw them
    away
195                                 # wait for the next batch to come in to see if the
196                                 # twin is there
197                                 print("did i get here?")
198                                 pass
199
200                         ####################
201                         # PROCESS MESSAGES #
202                         ####################
203                         # tell a thread to handle the scoring of the current
    patients so far
204
205                         ##################
206                         # CLOSE MESSAGES #
207                         ##################
208                         else:
209                             # should be the CLOSE Message
210                             # close all files
211                             print("closing instance loader thread...")
212                             if self.__file is not None:
213                                 self.__file.close()
214                             __running = False
215                             break
216
217                         # always remove message after using it
218                         self.__raw_instances.pop(0)
219
220                 else:
221                     time.sleep(1)
222
223         finally:
224
225             if self.__file is not None:
226                 self.__file.close()
227
228
229 """
230 SERVER
231
232 code to run the actual server that communicates with the wearable
233 device via the XBee connected via USB serial port.
234 """
235
236
237 def run_server():
238     """
239     Xbee Server
240     :return:
241     """
242     # INIT SERVER
243     print("starting run_server...")
244     UpdaWear.reset()
245     MessageBuffer.clear()
246
247     server = Raw802Device(PORT, BAUD_RATE)
248     UpdaWear.id = "UPDA-WEAR-1"
249     UpdaWear.address = XBee16BitAddress.from_hex_string(WEAR_16B_ADDR)
250     UpdaWear.remote = RemoteRaw802Device(server, UpdaWear.address)
251     instance_manager = None
252     exit_message = bytearray([0x05, 0x00])
253
254     try:
255         # might not want this
```

```
256         if not os.path.exists('./data'):
257             print("unable to find './data' to store information")
258             raise KeyboardInterrupt
259 
260         # get data count to make a new file
261         num_patients = len([name for name in os.listdir('./data')])
262 
263         instance_manager = InstanceLoader(num_patients)
264         instance_manager.start()  # start the thread
265         server.open()
266 
267         def msg_callback(message):
268             UpdaWear.received_count = UpdaWear.received_count + 1
269             # register the device
270             # print("{} >> {}".format(server.get_16bit_addr(), message.data.decode
    ()))
271             # pass information off to a buffer
272             # store the data (byte array)
273             with BufferLock:
274                 MessageBuffer.append(message.data)
275 
276         server.add_data_received_callback(msg_callback)
277 
278         print("press enter to stop run_server...")
279         input()
280 
281     except KeyboardInterrupt:
282         print()  # add a space so everything is nearly on a different line
283 
284     except serial.serialutil.SerialException:
285         print("Unable to open {}".format(PORT))
286 
287     finally:
288 
289         if instance_manager is not None:
290             # close instance manager
291             with BufferLock:
292                 MessageBuffer.append(exit_message)
293             instance_manager.join()
294 
295         if server is not None and server.is_open():
296             server.close()
297 
298     print("closing run_server...")
299 
300 
301 """
302 Simple UPDA CONSOLE
303 
304 functions that are triggered based on input from the actual
305 console system.
306 """
307 
308 
309 def stats(tokens):
310     """
311     Present stats on the previous server session
312     :param tokens:
313     :return:
314     """
315     print("stats from previous run_server session...")
316     print("previously linked with: {}".format(UpdaWear.id))
317     print("address: {}".format(UpdaWear.address))
318     print("received messages: {}".format(UpdaWear.received_count))
319     print("known wearable runtime: {} seconds".format(0.01 * UpdaWear.received_count
    ))
320 
```

```
321
322 def start(tokens):
323     """
324     Initialize a new 'process'
325     start run_server
326     start process
327     start process data-1
328
329     :param tokens: arguments to specify what is being started
330     :return: NA
331     """
332     if len(tokens) <= 1:
333         print("no object specified...")
334         return
335
336     if tokens[1] == "server":
337         # start run_server procedure
338         run_server()
339
340     # command to explicitly start processing data
341     if tokens[1] == "process":
342
343         if len(tokens) > 2:
344             # specifying an object to process
345             patient_name = tokens[2]
346             print("process {}".format(patient_name))
347             # check if data exists
348             if not os.path.exists("./data/{}".format(patient_name)):
349                 print("error: unable to find ./data/{} to process".format(
    patient_name))
350                 return
351
352             # check if data has been processed
353             # if data has a pdf score, then it has been processed
354             if os.path.exists("./data/{}/UPDAReport.pdf".format(patient_name)):
355                 print("{} has already been processed".format(patient_name))
356                 return
357
358             # process
359             manager = PipelineManager(patient_path="./data/{}".format(patient_name))
360             manager.start()
361             manager.join()
362             print("{} processing complete".format(patient_name))
363
364         else:
365             # FEATURE TO COME
366             # just process the first (or all)
367             print("process all patients")
368             # get all patients
369             # check if data has been processed
370             # process
371             pass
372
373     else:
374         print("unknown object to start...")
375         print("options to run:\nstart server\nstart process\nstart process <data-
    name>")
376
377
378 def load(tokens):
379     """
380     :param tokens: arguments in the case of additional data
381     :return:
382     """
383     print("downloading data information from sd...")
384     if not os.path.isdir(SD_PATH):
385         # cannot find the SD card
```

```
386         print("SD card {} does not exist".format(SD_PATH))
387 
388     else:
389         # found the SD card
390         # check for the DATA file
391         print("SD card {} found".format(SD_PATH))
392         if not os.path.exists(SD_DATA_PATH):
393             print("cannot continue: {} not found".format(SD_DATA_PATH))
394         else:
395             # download file
396             with open(SD_DATA_PATH, 'r') as dfile:
397                 content = dfile.read()
398                 content = content.split(sep='----- datafile -----')
399 
400                 print("{} sets of data data found".format(len(content)-1))
401                 print("downloading data data")
402 
403                 bar_size = 50
404 
405                 if not (len(content)-1 <= 0):
406                     step = int(bar_size/(len(content)-1))
407 
408                     # how many data files exist so far
409                     num_patients = len([name for name in os.listdir('./data')])
410 
411                     # download files
412                     for i in range(1, len(content)):
413                         print('|','%'*(step*i), ' '*(bar_size - (step*i)), '|', end=
    '\r')
414                         # mahoney filters for each imu
415                         hand_filter = MahoneyFilter()
416                         thumb_filter = MahoneyFilter()
417                         point_filter = MahoneyFilter()
418                         ring_filter = MahoneyFilter()
419 
420                         try:
421                             lines = content[i].split(sep='\n')
422                             os.mkdir("./data/patient-{}".format(str(i+num_patients))
    )
423                             f = open("./data/patient-{}/raw.txt".format(str(i+
    num_patients)), 'w')
424                             pos = open("./data/patient-{}/pos.txt".format(str(i+
    num_patients)), 'w')
425 
426                             for line in lines:
427                                 # row = line.split(sep=' ')
428                                 row = line.split(sep=' ')
429                                 row_pos = []
430 
431                                 if len(row) != 38:
432                                     continue
433 
434                                 # hand IMU
435                                 hand_filter.process(float(row[2]), float(row[3]),
    float(row[4]), float(row[5]), float(row[6]), float(row[7]), float(row[8]), float(row
    [9]), float(row[10]), 0.01)
436                                 # hand IMU
437                                 thumb_filter.process(float(row[11]), float(row[12]),
     float(row[13]), float(row[14]), float(row[15]), float(row[16]), float(row[17]),
    float(row[18]), float(row[19]), 0.01)
438                                 # hand IMU
439                                 point_filter.process(float(row[20]), float(row[21]),
     float(row[22]), float(row[23]), float(row[24]), float(row[25]), float(row[26]),
    float(row[27]), float(row[28]), 0.01)
440                                 # hand IMU
441                                 ring_filter.process(float(row[29]), float(row[30]),
    float(row[31]), float(row[32]), float(row[33]), float(row[34]), float(row[35]),
```

```
441 float(row[36]), float(row[37]), 0.01)
442
443                                 row_pos.extend(hand_filter.q)
444                                 row_pos.extend(thumb_filter.q)
445                                 row_pos.extend(point_filter.q)
446                                 row_pos.extend(ring_filter.q)
447
448                                 row_pos.extend([hand_filter.to_pitch(), hand_filter.
    to_roll(), hand_filter.to_yaw()])
449                                 row_pos.extend([thumb_filter.to_pitch(),
    thumb_filter.to_roll(), thumb_filter.to_yaw()])
450                                 row_pos.extend([point_filter.to_pitch(),
    point_filter.to_roll(), point_filter.to_yaw()])
451                                 row_pos.extend([ring_filter.to_pitch(), ring_filter.
    to_roll(), ring_filter.to_yaw()])
452
453                                 # writing items
454                                 for item in row:
455                                     f.write("{} ".format(str(float(item))))
456                                 f.write('\n')
457
458                                 for item in row_pos:
459                                     pos.write("{} ".format(str(float(item))))
460                                 pos.write('\n')
461
462                             # f.write(content[i])
463                             f.close()
464
465                             # calculate the mahoney filter of each
466
467                         except BlockingIOError:
468                             print("\nerror: could not open new data file")
469                             break
470
471                     else:
472                         print("\ndownload complete!")
473
474
475 def list_items(tokens):
476     """
477     :param tokens:
478     :return:
479     """
480     if len(tokens) <= 1:
481         print("specify an object to list:")
482         print("patients")
483         return
484
485     if tokens[1] == 'patients':
486         if os.path.exists('./data'):
487             patients = [name for name in os.listdir('./data')]
488             print("number of data records on disk: {}".format(len(patients)))
489             # list all the files
490             print("./data:")
491             for i in patients:
492                 print(i)
493         else:
494             print("cannot find path './data'")
495
496     else:
497         print("unknown object to list...")
498
499
500 def test_module(tokens):
501     """
502
503     :param tokens:
```

```
504     :return:
505     """
506     print("Testing Module")
507     matrix = extract('./data/smallset', 'HAx')
508
```

```
#! /usr/bin/python

"""
[SERVER] Unsupervised Parkinson's Disease Assessment

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

Console class
"""
from server import *


"""
[CLASS] Command

Defines a new command a user can install into the console class
"""


class Command(object):

    def __init__(self, name, help_msg, action):
        self.__name = name
        self.__help = help_msg
        self.__action = action

    def name(self):
        """
        :return: (str) name of the command
        """
        return self.__name

    def __call__(self, tokens):
        """
        :param tokens: (list) : arguments from user input that can be used to run the command
        :return:
        """
        self.__action(tokens)

    def __str__(self):
        """
        :return: (str) : string formatted description of what is shown in the "help" menu
        """
        return self.__name + ('\t' * 3) + self.__help


"""
[CLASS] Console

Defines a generic class for designing a basic console interface for the application.
New command objects can be created and added to the console class to enable different
types of features.
"""


class Console(object):

    def __init__(self, name):
        self.__name = name
        self.__prompt = "> "
        self.__separator = ' '
        self.__closer = "goodbye"
        # list of command objects
        self.__commands = dict()
```

```
 66         self.__commands['help'] = Command(name="help", help_msg="description of how
    to use the commands", action=self.cmd_help)
 67         self.__commands['exit'] = Command(name='exit', help_msg='exits the console',
     action=self.cmd_exit)
 68         # control objects
 69         self.__run = True
 70 
 71     def set_prompt(self, prompt):
 72         """
 73         :param prompt: (str) : prompt to set
 74         :return: NA
 75         """
 76         self.__prompt = prompt
 77 
 78     def set_separator(self, sep):
 79         """
 80         :param sep: (str): seperator  to set
 81         :return: NA
 82         """
 83         self.__separator = sep
 84 
 85     def set_closer(self, close):
 86         """
 87         :param close: (str): set an exit message
 88         :return: NA
 89         """
 90 
 91         self.__closer = close
 92 
 93     def add_command(self, command):
 94         """
 95         :param command: (Command) : command to be added to console
 96         :return: NA
 97         """
 98         if not type(command) is Command:
 99             print("invalid command to add")
100             return
101 
102         if not command.name() in self.__commands:
103             self.__commands[command.name()] = command
104 
105     def cmd_help(self, tokens):
106         """
107         Function for the help command
108         :param tokens: (list): arguments to pass to the command
109         :return:  NA
110         """
111 
112         print("usage: {}\n".format(self.__name))
113         for cmd in self.__commands.items():
114             print(cmd[1])
115 
116     def cmd_exit(self, tokens):
117         """
118         Signal the thread to terminate the console
119         Called upon the "exit" command
120         :param tokens: arguments
121         :return:
122         """
123         self.__run = False
124 
125     def run(self):
126         """
127         Main loop that drives the console
128         :return:
129         """
130         try:
```

```
131             while self.__run:
132                 tokens = input(self.__prompt).lower().split(sep=self.__separator)
133                 # print(tokens)
134                 if tokens[0] in self.__commands:
135                     # running command
136                     self.__commands[tokens[0]](tokens)
137 
138                 else:
139                     print("unknown cmd '{}'".format(tokens[0]))
140 
141                 print()
142 
143         except KeyboardInterrupt:
144             pass
145 
146         finally:
147             print(self.__closer)
148 
149 
150 def main():
151     """
152     The function that defines the console and starts the entire
153     software system on the server
154     :return:
155     """
156     print('*' * 50)
157     print("\n  .----.-----.-----.-----.\n",
158           "/      \     \     \     \\\n",
159           "|  \/    |     |   __L_____L__\n",
160           "|   |    |     |  (           \\\n",
161           "|    \___/    /    \_____/    |\n",
162           "|        \___/\___/\___/       |\n",
163           "|      \     /               /\n",
164           "|                         __/\n",
165           "\_                      __/\n",
166           "|        |         |\n",
167           "|                  |\n",
168           "|                  |\n\n")
169 
170     print("UPDA Console")
171     print("Version {}".format(VERSION), '\n')
172     print('*' * 50, '\n')
173 
174     console = Console("UPDA Console")
175 
176     # generate all commands
177     start_cmd = Command(name='start', help_msg='start a subroutine (start server|
    process)', action=start)
178     stats_cmd = Command(name='stat', help_msg='produce statistics about the previous
     server runtime', action=stats)
179     load_cmd = Command(name='load', help_msg='load data from the sd card and write
    to the server', action=load)
180     list_cmd = Command(name='list', help_msg='list specified items. ex: list
    patients', action=list_items)
181     test_cmd = Command(name='test', help_msg='command to easily test new code
    features', action=test_module)
182 
183     # add commands
184     console.add_command(start_cmd)
185     console.add_command(stats_cmd)
186     console.add_command(load_cmd)
187     console.add_command(list_cmd)
188     console.add_command(test_cmd)
189 
190     # console.run()
191     console.run()
192 
```

```
193
194 if __name__ == '__main__':
195     main()
196
```

```
"""
[CLASS] Reporter

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

 Responsible for generating a presentable report for researchers
 and medical professionals. Ideally in the future this can be used
 to effectively diagnose a patients condition
"""
import os
import datetime
from PyPDF2 import PdfFileReader, PdfFileWriter
from reportlab.pdfgen import canvas


class Reporter(object):

    POINT = 1
    INCH = 72
    FILENAME = "score.pdf"
    REPORT = "UPDAReport.pdf"
    GRAPH = "graph.pdf"

    def __init__(self, filepath):
        self.__patient_path = filepath

    def generate_report(self, score):
        """

        :param score:
        :return:
        """
        self.__generate_score(score=score)
        self.__merge_reports(patient_path=self.__patient_path)
        os.remove("{}/{}".format(self.__patient_path, self.FILENAME))
        print("UPDAReport Complete!")

    def __generate_score(self, score):
        """
        :param patient_path:
        ex: ./data/data-1
        Generate a PDF with the score of the patient's tremors for the day
        :return:
        """

        if not os.path.exists(path=self.__patient_path):
            print("error: {} does not exist, cannot generate report".format(self.
__patient_path))
            return

        c = canvas.Canvas("{}/score.pdf".format(self.__patient_path), pagesize=(8.5*
self.INCH, 11*self.INCH))
        c.setStrokeColorRGB(0, 0, 0)
        c.setFillColorRGB(0, 0, 0)
        c.setFont("Helvetica", 12 * self.POINT)

        # generate data name
        v = 10 * self.INCH
        c.drawString(7 * self.INCH, v, score['name'])
        # generate datetime
        v = 10 * self.INCH
        v -= 12 * 4 * self.POINT
        c.drawString(3.25 * self.INCH, v, str(datetime.datetime.now()))
        # generate finger tap scores
        v = 10 * self.INCH
        v -= 12 * 12.5 * self.POINT
```

```
66          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['ftap'][0]))
67          c.drawString(6.40 * self.INCH, v, "{}".format(score['ftap'][1]))
68          # generate hand movement scores
69          v = 10 * self.INCH
70          v -= 12 * 14.75 * self.POINT
71          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['htap'][0]))
72          c.drawString(6.40 * self.INCH, v, "{}".format(score['htap'][1]))
73          # generate postural tremor scores
74          v = 10 * self.INCH
75          v -= 12 * 17 * self.POINT
76          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['ptrem'][0]))
77          c.drawString(6.40 * self.INCH, v, "{}".format(score['ptrem'][1]))
78          # generate kinetic tremor scores
79          v = 10 * self.INCH
80          v -= 12 * 19.25 * self.POINT
81          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['ktrem'][0]))
82          c.drawString(6.40 * self.INCH, v, "{}".format(score['ktrem'][1]))
83          # generate rest tremor scores
84          v = 10 * self.INCH
85          v -= 12 * 21.5 * self.POINT
86          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['rtrem'][0]))
87          c.drawString(6.40 * self.INCH, v, "{}".format(score['rtrem'][1]))
88          # generate consistency of rest scores
89          v = 10 * self.INCH
90          v -= 12 * 23.75 * self.POINT
91          # c.drawString(4.25 * self.INCH, v, "{}%".format(score['crest'][0]))
92          c.drawString(6.40 * self.INCH, v, "{}".format(score['crest'][1]))
93
94          c.showPage()
95          c.save()
96
97      def __merge_reports(self, patient_path):
98          """
99          :param patient_path:
100         ex: ./data/data-1
101         Combine scoring PDF with UPDA template to make it look nicer
102         :return:
103         """
104         output = PdfFileWriter()
105
106         head = PdfFileReader(open("./resources/HeadScore.pdf", "rb"))
107         head_page = head.getPage(0)
108
109         score = PdfFileReader(open("{}/{}".format(patient_path, self.FILENAME), "rb"
    ))
110         head_page.mergePage(score.getPage(0))
111
112         output.addPage(head_page)
113         output_stream = open("{}/{}".format(patient_path, self.REPORT), "wb")
114         output.write(output_stream)
115
```

```
"""
MATRIX BUILDER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

Builds matrices for all the filters that need to extract information from all the
other different files.

SYSTEM
in function extract, you specify what values to extract.
there is a code to specify what you want to extract
the order that the values come specifies what order the values are in
"""
from yaml import load

"""
CLASS + SUBCLASS + ITEM
 (H)      (G)       (x)

example:
3
extract('data-1', 'HA', 'TGx', 'TGy', 'M')

"""


def extract(filename, *argv):
    """
    :param filename: the name of the patient to extract data from
    :param argv: strings that have requests for what data to collect.
                 all different types of data available to collect are viewable in
                 "server/resources/data_sources.yaml".
                 Please look at the example above for how to use this function
    :return:
    """
    yamlfile = load(open('./resources/data_sources.yaml', 'r'))

    result = []

    # iterating through arguments
    for arg in argv:
        # ex: HAx
        # load file
        indices = []  # all items to retrieve

        # print('class: {}'.format(arg[0]))
        # print('file path: {}'.format(yamlfile[arg[0]]['file_path']))

        if len(arg) == 1:
            # select the entire range
            # print('range: {}'.format(yamlfile[arg[0]]['range']))
            indices = yamlfile[arg[0]]['range']

        if len(arg) == 2:
            # select the subclass range
            # print('subclass: {}'.format(arg[1]))
            # print('range: {}'.format(yamlfile[arg[0]][arg[1]]['range']))
            indices = yamlfile[arg[0]][arg[1]]['range']

        if len(arg) == 3:
            # select selector location
            # print('subclass: {}'.format(arg[1]))
            # print('selector: {}'.format(arg[2]))
            # print('location: {}'.format(yamlfile[arg[0]][arg[1]][arg[2]]))
            indices = yamlfile[arg[0]][arg[1]][arg[2]]

```

```
        # print()
        # print(indices)

        # initializing result values
        if len(indices) == 1:
            # just a location
            values = [[]]
        else:
            # a range of values
            values = [[] for i in range(indices[0], indices[1])]

        # print(values)
        # print()

        #################
        # Enter in Data #
        #################
        datafile = None

        try:
            # open the raw file, lowpass file, etc
            datafile = open('{}/{}'.format(filename, yamlfile[arg[0]]['file_path']))
            # extract all values and put in matrix
            if len(indices) == 1:
                # just one location
                for line in datafile:
                    values[0].append(float(line.split(sep=' ')[indices[0]]))

            else:
                # range of locations
                for row in datafile:
                    line = row.split(sep=' ')
                    for i in range(0, indices[1] - indices[0]):
                        values[i].append(float(line[i+indices[0]]))

            # add on lists
            result.extend(values)

        finally:
            datafile.close()

    return result

```

```
"""
PIPELINE MANAGER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

After the raw data has been produced by the server, the server starts a new
thread called the pipeline manager, which guides the raw data through
several filters and ultimately to the scoring stage, where the system
will produce the UPDRS results
"""
import time
import os
from analysis.LowPassFilter import LowPassFilter
from analysis.BandPassFilter import BandPassFilter
from analysis.HampelFilter import HampelFilter
from analysis.GravityFilter import GravityFilter
from Reporter import Reporter
from Score import Score
from threading import Thread, Lock, ThreadError


"""
[CLASS] PipelineManager

This derivation of the thread class is responsible for passing raw data from
a patient profile through all of the different filters in the pipeline.
"""


class PipelineManager(Thread):

    def __init__(self, patient_path):
        """
        :param patient_path: name of the folder containing all the information about
the data
        example: ./data/data-1

        """
        Thread.__init__(self)
        self.__patient_path = patient_path

        self.__low_pass_filter = LowPassFilter(filename=self.__patient_path)
        self.__band_pass_filter = BandPassFilter(filename=self.__patient_path)
        self.__hampel_filter = HampelFilter(filename=self.__patient_path)
        self.__gravity_filter = GravityFilter(filename=self.__patient_path)
        self.__score = Score(filename=self.__patient_path)
        self.__reporter = Reporter(filepath=self.__patient_path)

    def run(self):
        """
        Given a data profile, pass that data's profile through all the
        stages of the UPDA pipeline
        :return:
        """
        ##########
        # Checks #
        ##########
        if not os.path.exists("{}".format(self.__patient_path)):
            print("error: data does not exist")
            return

        if not os.path.exists("{}/raw.txt".format(self.__patient_path)):
            print("error: data does not exist")
            return

        print("processing: {}".format(self.__patient_path))
```

```
 67         start = time.time()
 68
 69         ####################
 70         # Low Pass Filter #
 71         ####################
 72         print("calling low pass filter...")
 73         self.__low_pass_filter.process()
 74
 75         ####################
 76         # Band Pass Filter #
 77         ####################
 78         print("calling band pass filter...")
 79         self.__band_pass_filter.process()
 80
 81         ####################
 82         # Hampel Filter    #
 83         ####################
 84         print("calling hampel filter...")
 85         self.__hampel_filter.process()
 86
 87         ####################
 88         # Gravity Filter   #
 89         ####################
 90         print("calling gravity filter...")
 91         self.__gravity_filter.process()
 92
 93         ##################
 94         # Scoring Filter #
 95         ##################
 96         print("calling score...")
 97         self.__score.process()
 98
 99         ##################
100         # Report Output  #
101         ##################
102         print("reporting...")
103         self.__reporter.generate_report(self.__score.get_result())
104
105         print("Processing Time: {}".format(time.time() - start))
106
```

```
"""
[CLASS] HAMPEL FILTER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly, Senbao Lu


"""
import pandas as pd
import numpy as np
from MatrixBuilder import extract
from math import *


class HampelFilter(object):

    AVE_BOUND = 1

    def __init__(self, filename):
        self.__filename = filename

    def process(self):
        """

        emg rectified
        """
        emg_rekt = extract(self.__filename, 'EC')

        output = open("{}/hampel.txt".format(self.__filename), "w")
        filtered = self.hampel(vals_orig=emg_rekt[0])

        for i in range(1, len(filtered)-1):
            count = 0
            total = 0
            top_cutoff = False

            # searching for nans
            if np.isnan(filtered[i]):
                while np.isnan(filtered[i+count]):
                    count += 1
                    if i + count >= len(filtered) - 1:
                        top_cutoff = True
                        break

                # lower bounds of nans
                for r in range(i-self.AVE_BOUND, i):
                    total += filtered[r]

                if not top_cutoff:
                    # upper bound of nans
                    for r in range(i+count, i+count+self.AVE_BOUND):
                        total += filtered[r]

                for n in range(i, i+count):
                    if not top_cutoff:
                        filtered[n] = total/(2.0 * self.AVE_BOUND)

                    else:
                        filtered[n] = total / self.AVE_BOUND

                if top_cutoff:
                    filtered[len(filtered)-1] = filtered[n] = total / self.AVE_BOUND
                    break

                # maximum = max(filtered[i-self.AVE_BOUND: i].extend(filtered[i+count
: i+count+self.AVE_BOUND]))

```

```
 67                 # for n in range(i, i+count):
 68                 #     filtered[nan] = maximum
 69
 70         try:
 71             for val in filtered:
 72                 output.write(str(val))
 73                 output.write("\n")
 74
 75         finally:
 76             output.close()
 77
 78     def hampel(self, vals_orig, k=17, t0=3):
 79         """
 80         credit goes to: https://ocefpaf.github.io/python4oceanographers/blog/2015/03
    /16/outlier_detection/
 81         https://stackoverflow.com/questions/46819260/filtering-outliers-how-to-make-
    median-based-hampel-function-faster
 82
 83         :param vals_orig: pandas series of values from which to remove outliers
 84         :param k: size of the window (including the sample; 7 is equal to 3 on
    either side of value)
 85         :param t0:
 86         :return:
 87         """
 88         # make copy so original not edited
 89         vals = pd.Series(vals_orig.copy())
 90
 91         # hampel filter
 92         L = 1.4826
 93         rolling_median = vals.rolling(k).median()
 94         difference = np.abs(rolling_median-vals)
 95         median_abs_deviation = difference.rolling(k).median()
 96         threshold = t0 * L * median_abs_deviation
 97         outlier_idx = difference > threshold
 98
 99         vals[outlier_idx] = np.nan
100         return vals
101
```

```
"""
[CLASS] MAHONEY FILTER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

A type of kallman filter that fuses acceleration, radians per second,
microtesslas, and a time differential to estimate the position of
the imu in space. This is then recorded into the

credits for the math:
https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles
"""
from math import *


class MahoneyFilter(object):

    Kp = 2.0 * 5.0
    Ki = 0.0
    GyroMeasError = pi * (40.0 / 180.0)
    GyroMeasDrift = pi * (0.0 / 180.0)
    Beta = sqrt(3.0 / 4.0) * GyroMeasError
    Zeta = sqrt(3.0 / 4.0) * GyroMeasDrift

    def __init__(self):
        self.q = [1.0, 0.0, 0.0, 0.0]  # w x y z
        self.eint = [0.0, 0.0, 0.0]

    def process(self, ax, ay, az, gx, gy, gz, mx, my, mz, dt):
        """
        instance:

        emg_raw emg_rect Hand_Axyz_Gxyz_Mxyz Thumb_Axyz_Gxyz_Mxyz
Pointer_Axyz_Gxyz_Mxyz Ring_Axyz_Gxyz_Mxyz

        one instance consists of:
        Hand_Axyz_Gxyz_Mxyz

        :param ax: acceleration in the x-direction
        :param ay: acceleration in the y-direction
        :param az: acceleration in the z-direction
        :param gx: radians/second in the x-direction
        :param gy: radians/second in the y-direction
        :param gz: radians/second in the z-direction
        :param mx: micro tesslas in the x direction
        :param my: micro tesslas in the y direction
        :param mz: micro tesslas in the z direction
        :param dt: time in seconds since the last update
        :return: None
        Information is stored in 'self._q' for future work
        """
        q1 = self.q[0]
        q2 = self.q[1]
        q3 = self.q[2]
        q4 = self.q[3]

        q1q1 = q1 * q1
        q1q2 = q1 * q2
        q1q3 = q1 * q3
        q1q4 = q1 * q4
        q2q2 = q2 * q2
        q2q3 = q2 * q3
        q2q4 = q2 * q4
        q3q3 = q3 * q3
        q3q4 = q3 * q4
        q4q4 = q4 * q4
```

```
 67
 68         # normalize accel
 69         norm = sqrt(ax * ax + ay * ay + az * az)
 70         if norm == 0.0:
 71             return
 72         norm = 1.0 / norm
 73         ax *= norm
 74         ay *= norm
 75         az *= norm
 76
 77         # normalize mag
 78         norm = sqrt(mx * mx + my * my + mz * mz)
 79         if norm == 0.0:
 80             return
 81         norm = 1.0 / norm
 82         mx *= norm
 83         my *= norm
 84         mz *= norm
 85
 86         # reference direction of magnetic field
 87         hx = 2.0 * mx * (0.5 - q3q3 - q4q4) + 2.0 * my * (q2q3 - q1q4) + 2.0 * mz *
    (q2q4 + q1q3)
 88         hy = 2.0 * mx * (q2q3 + q1q4) + 2.0 * my * (0.5 - q2q2 - q4q4) + 2.0 * mz *
    (q3q4 - q1q2)
 89         bx = sqrt((hx * hx) + (hy * hy))
 90         bz = 2.0 * mx * (q2q4 - q1q3) + 2.0 * my * (q3q4 + q1q2) + 2.0 * mz * (0.5 -
     q2q2 - q3q3)
 91
 92         # estimated direction of gravity and magnetic field
 93         vx = 2.0 * (q2q4 - q1q3)
 94         vy = 2.0 * (q1q2 + q3q4)
 95         vz = q1q1 - q2q2 - q3q3 + q4q4
 96         wx = 2.0 * bx * (0.5 - q3q3 - q4q4) + 2.0 * bz * (q2q4 - q1q3)
 97         wy = 2.0 * bx * (q2q3 - q1q4) + 2.0 * bz * (q1q2 + q3q4)
 98         wz = 2.0 * bx * (q1q3 + q2q4) + 2.0 * bz * (0.5 - q2q2 - q3q3)
 99
100         # error is cross product between estimated direction and measured direction
    of gravity
101         ex = (ay * vz - az * vy) + (my * wz - mz * wy)
102         ey = (az * vx - ax * vz) + (mz * wx - mx * wz)
103         ez = (ax * vy - ay * vx) + (mx * wy - my * wx)
104
105         if self.Ki > 0.0:
106             self.eint[0] += ex
107             self.eint[1] += ey
108             self.eint[2] += ez
109
110         else:
111             # prevent integral windup
112             self.eint[0] = 0.0
113             self.eint[1] = 0.0
114             self.eint[2] = 0.0
115
116         # apply feedback terms
117         gx = gx + self.Kp * ex + self.Ki * self.eint[0]
118         gy = gy + self.Kp * ey + self.Ki * self.eint[1]
119         gz = gz + self.Kp * ez + self.Ki * self.eint[2]
120
121         # integrate rate of change of quaternion
122         pa = q2
123         pb = q3
124         pc = q4
125         q1 = q1 + (-q2 * gx - q3 * gy - q4 * gz) * (0.5 * dt)
126         q2 = pa + (q1 * gx + pb * gz - pc * gy) * (0.5 * dt)
127         q3 = pb + (q1 * gy - pa * gz + pc * gx) * (0.5 * dt)
128         q4 = pc + (q1 * gz + pa * gy - pb * gx) * (0.5 * dt)
129
```

```
130         # normalize quaternion
131         norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4)
132         norm = 1.0 / norm
133         self.q[0] = q1 * norm
134         self.q[1] = q2 * norm
135         self.q[2] = q3 * norm
136         self.q[3] = q4 * norm
137 
138     def to_roll(self, deg=False):
139         """
140         Converts the current Q into roll
141 
142         :param deg: (bool) should it be converted to degrees
143         :return: current estimated roll position in either degrees or radians
144         """
145         t0 = 2.0 * (self.q[0] * self.q[1] + self.q[2] * self.q[3])
146         t1 = self.q[0] * self.q[0] - self.q[1] * self.q[1] - self.q[2] * self.q[2] +
     self.q[3] * self.q[3]
147 
148         if deg:
149             return degrees(atan2(t0, t1))
150 
151         else:
152             return atan2(t0, t1)
153 
154     def to_pitch(self, deg=False):
155         """
156         Converts the current Q into pitch
157 
158         :param deg: (bool) should it be converted to degrees
159         :return: current estimated pitch position in either degrees or radians
160         """
161         t0 = 2.0 * (self.q[1] * self.q[3] - self.q[0] * self.q[2])
162 
163         if deg:
164             return degrees(-1.0 * asin(t0)) - 8.5  # according to sparkfun code.
    declination angle
165 
166         else:
167             return asin(-1.0 * t0)
168 
169     def to_yaw(self, deg=False):
170         """
171         Converts the current Q into yaw
172 
173         :param deg: (bool) should it be converted to degrees
174         :return: current estimated yaw position in either degrees or radians
175         """
176         t0 = 2.0 * (self.q[1] * self.q[2] + self.q[0] * self.q[3])
177         t1 = self.q[0] * self.q[0] + self.q[1] * self.q[1] - self.q[2] * self.q[2] -
     self.q[3] * self.q[3]
178 
179         if deg:
180             return degrees(atan2(t0, t1))
181 
182         else:
183             return atan2(t0, t1)
184 
185 
186 # calculations outside of the class
187 def q_to_roll(q, deg=False):
188     """
189     Converts the current Q into roll
190 
191     :param: q: (list) contains quaternion coordinates
192     :param deg: (bool) should it be converted to degrees
193     :return: current estimated roll position in either degrees or radians
```

```
194     """
195     t0 = 2.0 * (q[0] * q[1] + q[2] * q[3])
196     t1 = q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] * q[3]
197 
198     if deg:
199         return degrees(atan2(t0, t1))
200 
201     else:
202         return atan2(t0, t1)
203 
204 
205 def q_to_pitch(q, deg=False):
206     """
207     Converts the current Q into pitch
208 
209     :param: q: (list) contains quaternion coordinates
210     :param deg: (bool) should it be converted to degrees
211     :return: current estimated pitch position in either degrees or radians
212     """
213     t0 = 2.0 * (q[1] * q[3] - q[0] * q[2])
214 
215     if deg:
216         return degrees(-1.0 * asin(t0)) - 8.5  # according to sparkfun code.
    declination angle
217 
218     else:
219         return asin(-1.0 * t0)
220 
221 
222 def q_to_yaw(q, deg=False):
223     """
224     Converts the current Q into yaw
225 
226     :param: q: (list) contains quaternion coordinates
227     :param deg: (bool) should it be converted to degrees
228     :return: current estimated yaw position in either degrees or radians
229     """
230     t0 = 2.0 * (q[1] * q[2] + q[0] * q[3])
231     t1 = q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] * q[3]
232 
233     if deg:
234         return degrees(atan2(t0, t1))
235 
236     else:
237         return atan2(t0, t1)
```

```
"""
[CLASS] GRAVITY FILTER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly, Senbao Lu

A filter that can find the direction of gravity given the orientation of the hand. This
direction can be used to discern when a data is resting their hand or they are
using their hand

RM_hand=[2.*(hqi.*hqk+hqj.*hqr).*g,2.*s.*(hqj.*hqk-hqi.*hqr).*g,(hqr.^2-hqi.^2-hqj.^2+hqk.^2).*g];

b = i
a = r
c = j
d = k

2(bd+ac) * g
2(cd - ab) * g
(a^2 - b^2 - c^2 + d^2) * g

"""
from math import *
import numpy as np
from MatrixBuilder import extract


class GravityFilter(object):

    GRAVITY = -9.81

    def __init__(self, filename):
        self.__filename = filename
        pass

    def process(self):
        # go to raw file with quads
        outfile = open("{}/gravity.txt".format(self.__filename), "w")

        def calc_accel(q, a):
            """
            :param q: [r, i, j, k] : quaternions
            :param a: [x, y, z] : raw acceleration
            :return: (list) "true" acceleration
            """
            x_comp = 2.0 * (q[0, 1] * q[0, 3] + q[0, 0] * q[0, 2]) * self.GRAVITY
            y_comp = 2.0 * (q[0, 2] * q[0, 3] - q[0, 0] * q[0, 1]) * self.GRAVITY
            z_comp = (pow(q[0, 0], 2.0) - pow(q[0, 1], 2.0) - pow(q[0, 2], 2.0) + pow(q[0, 3], 2.0)) * self.GRAVITY
            return [x_comp + a[0, 0], y_comp + a[0, 1], z_comp + a[0, 2]]

        rawAccel = extract(self.__filename, 'HA', 'TA', 'PA', 'RA', 'Q')
        rawAccelMat = np.matrix(rawAccel).transpose()

        try:
            for line in range(0, len(rawAccel[0])):

                # Ha = [0, 2]
                # Ta = [3, 5]
                # Pa = [6, 8]
                # Ra = [9, 11]
                # qh = [12, 15]
                # qt = [16, 19]
                # qp = [20, 23]
                # qr = [24, 27]
```

```
65
66                  qh = rawAccelMat[line, 12:16]
67                  qt = rawAccelMat[line, 16:20]
68                  qp = rawAccelMat[line, 20:24]
69                  qr = rawAccelMat[line, 24:28]
70
71                  # generate rotation matrix
72                  true_h = calc_accel(qh, rawAccelMat[line, 0:3])
73                  true_t = calc_accel(qt, rawAccelMat[line, 3:6])
74                  true_p = calc_accel(qp, rawAccelMat[line, 6:9])
75                  true_r = calc_accel(qr, rawAccelMat[line, 9:12])
76
77                  # store scores
78                  # HAND
79                  # print(rr_h.shape)
80                  # FOR EACH LINE:
81                  # HAx HAy HAz
82                  outfile.write("{} {} {} ".format(true_h[0], true_h[1], true_h[2]))
83                  # THUMB
84                  outfile.write("{} {} {} ".format(true_t[0], true_t[1], true_t[2]))
85                  # POINT
86                  outfile.write("{} {} {} ".format(true_p[0], true_p[1], true_p[2]))
87                  # RING
88                  outfile.write("{} {} {}\n".format(true_r[0], true_r[1], true_r[2]))
89
90          finally:
91              # print("!!! AFTER !!!")
92              outfile.close()
93
```

```
"""
[CLASS] LOW PASS FILTER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly, Senbao Lu

This signal filter will take the raw data and run it all through
a special low-pass filter.
"""
from scipy.signal import *
import numpy as np
import os
from MatrixBuilder import extract


class LowPassFilter(object):

    COEFFICIENT = str(os.getcwd()) + "/resources/lowpass_coef.txt"

    def __init__(self, filename):
        """
        filtering:
        2* emg  12 * accelerometer, 12 * gyroscope, 16 * positional

        [1 x 341] matrix
        :param filename: data name
        """
        self.__filename = filename
        self.__indicies = []
        self.__mat = list

    def process(self):
        # print(os.getcwd())
        rawfile = open("{}/raw.txt".format(self.__filename), "r")
        coefficients = open(self.COEFFICIENT, "r").read().split(sep='\n')
        coefficients = np.matrix(coefficients)
        output = open("{}/lowpass.txt".format(self.__filename), "w")

        self.__mat = extract(self.__filename, "E", "HA", "HG", "TA", "TG", "PA", "PG"
, "RA", "RG", "Q")
        self.__mat = np.matrix(self.__mat).transpose()

        convolution = convolve(self.__mat.astype(np.float64), coefficients.astype(np.
float64), mode='same').transpose()
        num_rows, num_cols = convolution.shape

        try:
            for c in range(0, num_cols):
                for r in range(0, num_rows):
                    output.write("{} ".format(convolution[r][c]))
                output.write('\n')

        finally:
            output.close()

```

```
"""
[CLASS] RawDataFilter

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly

Filters that process and use the
packet data to produce and anlaysis
"""
from analysis.MahonyFilter import MahoneyFilter


class RawDataFilter(object):

    BROADCAST_MSG = 1    # initializing communication
    NEW_DATASEG_MSG = 2  # new data file should be made
    OLD_DATASEG_MSG = 3  # previous data file was not finished, should be continued
    PAYLOAD_MSG = 4      # message containing data to store
    CLOSE_MSG = 5        # wearable telling server to stop
    PROCESS_MSG = 6      # wearable telling server to process the messages

    def __init__(self):
        self.hand_mhf = MahoneyFilter()
        self.thumb_mhf = MahoneyFilter()
        self.point_mhf = MahoneyFilter()
        self.ring_mhf = MahoneyFilter()

    def process(self, raw0, raw1):
        """
        extracts data from the two packets that represent one sampling instance.
        this information is then used to calculate the quaternion coordinate system,
        which is added and returned to the caller as one complete packet, a list of
numbers

        payload packet format should be: operation id emg_raw0 emg_raw1 ....

        :param raw0: first packet for instance
        :param raw1: second packet for instance
        :return:

        MESSAGE_TYPE, PAYLOAD_ID, PAYLOAD
        """

        # get command and id
        _command, _pid = int(str(raw0[0]), 16), int(str(raw0[1]), 16)

        # process command
        if _command == self.PAYLOAD_MSG:
            result = list()

            # process the payload
            # 1. extract all information
            # 2. calculate quaternion coordinates and roll, pitch, and yaw --> add to
 payload
            # 3. pass back to thread for storage

            ########################################
            # CONVERT BYTES TO FLOATS AND HALFWORDS #
            ########################################

            # lets just do EMG for testing purposes
            for i in range(2, 6, 2):
                result.append((raw0[i] << 8) + raw0[i+1])  # store RAW and RECT emg

            # build first half of floats
            # two bytes leftover here
            for i in range(6, 90, 4):
```

```
 66                 _real = (raw0[i] << 24) + (raw0[i+1] << 16) + (raw0[i+2] << 8) +
    raw0[i+3]
 67                 if _real > 0x7fffffff:
 68                     _real -= 0x100000000
 69                 result.append(float(_real)/100.0)
 70
 71             # build second half of floats
 72             # two bytes left over at index 2 and 3
 73             # total 100 bytes
 74             for i in range(2, 62, 4):
 75                 _real = (raw1[i] << 24) + (raw1[i+1] << 16) + (raw1[i+2] << 8) +
    raw1[i+3]
 76                 if _real > 0x7fffffff:
 77                     _real -= 0x100000000
 78                 result.append(float(_real)/100.0)
 79
 80             ##############################
 81             # Mahoney Filter Calculations #
 82             ##############################
 83             # emg values -> 0, 1
 84             # hand values -> 2 - 10
 85             # thumb values -> 11 - 19
 86             # pointer values -> 20 - 28
 87             # ring values -> 29 - 37
 88             self.hand_mhf.process(ax=result[2], ay=result[3], az=result[4], gx=
    result[5], gy=result[6], gz=result[7],
 89                                   mx=result[8], my=result[9], mz=result[10], dt=0.01
    )
 90
 91             self.thumb_mhf.process(ax=result[11], ay=result[12], az=result[13], gx=
    result[14], gy=result[15],
 92                                    gz=result[16], mx=result[17], my=result[18], mz=
    result[19], dt=0.01)
 93
 94             self.point_mhf.process(ax=result[20], ay=result[21], az=result[22], gx=
    result[23], gy=result[34],
 95                                    gz=result[25], mx=result[26], my=result[27], mz=
    result[28], dt=0.01)
 96
 97             self.ring_mhf.process(ax=result[29], ay=result[30], az=result[31], gx=
    result[32], gy=result[33],
 98                                   gz=result[34], mx=result[35], my=result[36], mz=
    result[37], dt=0.01)
 99
100             result.extend(self.hand_mhf.q)
101             result.extend(self.thumb_mhf.q)
102             result.extend(self.point_mhf.q)
103             result.extend(self.ring_mhf.q)
104             # return all info
105             return self.PAYLOAD_MSG, _pid, result
106
107         else:
108             # fragmentation error
109             # some sort of weird error...
110             print("error reading packet: {} - {}".format(_command, _pid))
111             return self.CLOSE_MSG, None, None
112
```

```
"""
[CLASS] BAND PASS FILTER

Date:       Tuesday June 12th, 2018
Author:     Alexander Adranly, Senbao Lu

This signal filter will take the raw data and run it all through
a special band-pass filter.
"""
import numpy as np
from scipy.signal import *
from MatrixBuilder import extract
import os


class BandPassFilter(object):

    COEFFICIENT = str(os.getcwd()) + "/resources/bandpass_coef.txt"
    FEATURE_SIZE = 28

    def __init__(self, filename):
        """
        filtering:
        12 * accelerometer, 12 * gyroscope, 4*roll

        [1 x 341] matrix
        :param filename:
        """
        self.__filename = filename
        self.__mat = list

    def process(self):
        # rawfile = open("{}/raw.txt".format(self.__filename), "r")
        coefficients = open(self.COEFFICIENT, "r").read().split(sep='\n')
        coefficients = np.matrix(coefficients)
        output = open("{}/bandpass.txt".format(self.__filename), "w+")

        self.__mat = extract(self.__filename, "HA", "HG", "TA", "TG", "PA", "PG", "RA
", "RG", "OHr", "OTr", "OPr", "ORr")
        self.__mat = np.transpose(self.__mat)

        # print("!!! BEFORE !!!")
        convolution = convolve(self.__mat.astype(np.float64), coefficients.astype(np.
float64), mode='same').transpose()
        num_rows, num_cols = convolution.shape
        # print("!!! AFTER !!!")

        try:
            for c in range(0, num_cols):
                for r in range(0, num_rows):
                    output.write("{} ".format(convolution[r][c]))
                output.write('\n')

        finally:
            output.close()

```

MatLab Code – Score

Table of Contents

read raw data

```
data=load("patient-1.txt");  % read the data
% INPUT FORMAT:
% EMG_raw EMG_rect
% Hand_Ax Hand_Ay Hand_Az Hand_Gx Hand_Gy Hand_Gz Hand_Mx Hand_My
 Hand_Mz
% Thumb_Ax Thumb_Ay Thumb_Az Thumb_Gx Thumb_Gy Thumb_Gz Thumb_Mx
 Thumb_My Thumb_Mz
% Point_Ax Point_Ay Point_Az Point_Gx Point_Gy Point_Gz Point_Mx
 Point_My Point_Mz
% Ring_Ax Ring_Ay Ring_Az Ring_Gx Ring_Gy Ring_Gz Ring_Mx Ring_My
 Ring_Mz
% Hand_Qr Hand_Qi Hand_Qj Hand_Qk
% Thumb_Qr Thumb_Qi Thumb_Qj Thumb_Qk
% Point_Qr Point_Qi Point_Qj Point_Qk
% Ring_Qr Ring_Qi Ring_Qj Ring_Qk
% Hand_Pp Hand_Py Hand_Pr
% Thumb_Pp Thumb_Py Thumb_Pr
% Point_Pp Point_Py Point_Pr
% Ring_Pp Ring_Py Ring_Pr
% EMG_hampel
fs=100;      %sampling frequency in HZ
[data_length,~]=size(data);
true_time=(1/fs:1/fs:data_length/fs)';
EMG_raw=data(:,1);    % EMG raw
EMG_rect=data(:,2);   % EMG rect
Hand_Ax=data(:,3);    % hand a
Hand_Ay=data(:,4);
Hand_Az=data(:,5);
Hand_Gx=data(:,6);    % hand g
Hand_Gy=data(:,7);
Hand_Gz=data(:,8);
Hand_Mx=data(:,9);    % hand m
Hand_My=data(:,10);
Hand_Mz=data(:,11);
Thumb_Ax=data(:,12);  % thumb a
Thumb_Ay=data(:,13);
Thumb_Az=data(:,14);
Thumb_Gx=data(:,15);  % thumb g
```

```
Thumb_Gy=data(:,16);
Thumb_Gz=data(:,17);
Thumb_Mx=data(:,18);  % thumb g
Thumb_My=data(:,19);
Thumb_Mz=data(:,20);
Point_Ax=data(:,21);  % point a
Point_Ay=data(:,22);
Point_Az=data(:,23);
Point_Gx=data(:,24);  % point g
Point_Gy=data(:,25);
Point_Gz=data(:,26);
Point_Mx=data(:,27);  % point m
Point_My=data(:,28);
Point_Mz=data(:,29);
Ring_Ax=data(:,30);   % ring a
Ring_Ay=data(:,31);
Ring_Az=data(:,32);
Ring_Gx=data(:,33);   % ring g
Ring_Gy=data(:,34);
Ring_Gz=data(:,35);
Ring_Mx=data(:,36);   % ring m
Ring_My=data(:,37);
Ring_Mz=data(:,38);
Hand_Qr=data(:,39);    % hand q
Hand_Qi=data(:,40);
Hand_Qj=data(:,41);
Hand_Qk=data(:,42);
Thumb_Qr=data(:,43);  % thumb q
Thumb_Qi=data(:,44);
Thumb_Qj=data(:,45);
Thumb_Qk=data(:,46);
Point_Qr=data(:,47);  % point q
Point_Qi=data(:,48);
Point_Qj=data(:,49);
Point_Qk=data(:,50);
Ring_Qr=data(:,51);   % ring q
Ring_Qi=data(:,52);
Ring_Qj=data(:,53);
Ring_Qk=data(:,54);
Hand_Pp=data(:,55);   % hand position
Hand_Py=data(:,56);
Hand_Pr=data(:,57);
Thumb_Pp=data(:,58);  % thumb position
Thumb_Py=data(:,59);
Thumb_Pr=data(:,60);
Point_Pp=data(:,61);  % point position
Point_Py=data(:,62);
Point_Pr=data(:,63);
Ring_Pp=data(:,64);   % ring position
Ring_Py=data(:,65);
Ring_Pr=data(:,66);
EMG_hampel=data(:,67);
```

Finger Taps

interruption ratio

```
interruption_ratio=interruption/taps_count;
if interruption_ratio<1
    interruption_rating=0;
elseif interruption_ratio<3
    interruption_rating=1;
elseif interruption_ratio<5
    interruption_rating=2;
elseif interruption_ratio<10
    interruption_rating=3;
else
    interruption_rating=4;
end
% speed
finger_tap_speed=finger_tap_time/taps_count;
if finger_tap_speed<1
    speed_rating=0;
elseif finger_tap_speed<3
    speed_rating=1;
elseif finger_tap_speed<5
    speed_rating=2;
elseif finger_tap_speed<10
    speed_rating=3;
else
    speed_rating=4;    % cannot perform: how to define
end
% amplitude decrement
decrement=5;
amplitude_rating=0;
if amplitude(2)-amplitude(1)>decrement
    amplitude_rating=3;
end
for i=2:7
    if amplitude(i+1)-amplitude(i)>decrement
        amplitude_rating=2;
    end
end
for i=8:9
    if amplitude(i+1)-amplitude(i)>decrement
        amplitude_rating=1;
    end
end
% Final Rating
finger_tap_rating=mode(interruption_rating,speed_rating,amplitude_rating);
disp(finger_tap_rating)
```

Hand Movements

interruption ratio

```
interruption_ratio=interruption/taps_count;
if interruption_ratio<1
    interruption_rating=0;
elseif interruption_ratio<3
    interruption_rating=1;
elseif interruption_ratio<5
    interruption_rating=2;
elseif interruption_ratio<10
    interruption_rating=3;
else
    interruption_rating=4;
end
% speed
finger_tap_speed=finger_tap_time/taps_count;
if finger_tap_speed<1
    speed_rating=0;
elseif finger_tap_speed<3
    speed_rating=1;
elseif finger_tap_speed<5
    speed_rating=2;
elseif finger_tap_speed<10
    speed_rating=3;
else
    speed_rating=4;    % cannot perform: how to define
end
% amplitude decrement
decrement=5;
amplitude_rating=0;
if amplitude(2)-amplitude(1)>decrement
    amplitude_rating=3;
end
for i=2:7
    if amplitude(i+1)-amplitude(i)>decrement
        amplitude_rating=2;
    end
end
for i=8:9
    if amplitude(i+1)-amplitude(i)>decrement
        amplitude_rating=1;
    end
end
% Final Rating
hand_movement_rating=mode(interruption_rating,speed_rating,amplitude_rating);
disp(hand_movement_rating)
```

Postural Tremor of Hands

calculate velocity from acceleration

```
Hand_Vx=zeros(data_length,1);
Hand_Vy=zeros(data_length,1);
Hand_Vz=zeros(data_length,1);
for i=2:data_length
```

```
    Hand_Vx(i,1)=Hand_Vx(i-1,1)+Hand_Ax(i-1,1)/fs;
end
for i=2:length(Hand_Vy)
    Hand_Vy(i,1)=Hand_Vy(i-1,1)+Hand_Ay(i-1,1)/fs;
end
for i=2:length(Hand_Vz)
    Hand_Vz(i,1)=Hand_Vz(i-1,1)+Hand_Az(i-1,1)/fs;
end
testing_time=zeros(data_length,1);
for i=26:data_length-25
    if Hand_Vx<0.1&&Hand_Vz<0.1&&Hand_Vy<0.1
        EMG=EMG_hampel(i-25:1:i+25,1);
        EMG_change=max(EMG)-min(EMG);
        if EMG_change>20
            testing_time(i,1)=1;
        end
    end
end
```

Kinetic Tremor of Hands

calculate velocity from acceleration

```
Hand_Vx=zeros(data_length,1);
Hand_Vy=zeros(data_length,1);
Hand_Vz=zeros(data_length,1);
for i=2:data_length
    Hand_Vx(i,1)=Hand_Vx(i-1,1)+Hand_Ax(i-1,1)/fs;
end
for i=2:length(Hand_Vy)
    Hand_Vy(i,1)=Hand_Vy(i-1,1)+Hand_Ay(i-1,1)/fs;
end
for i=2:length(Hand_Vz)
    Hand_Vz(i,1)=Hand_Vz(i-1,1)+Hand_Az(i-1,1)/fs;
end
testing_time=zeros(data_length,1);
for i=26:data_length-25
    if Hand_Vx<0.1&&Hand_Vz<0.1&&Hand_Vy>0.1
        EMG=EMG_hampel(i-25:1:i+25,1);
        EMG_change=max(EMG)-min(EMG);
        if EMG_change>20
            testing_time(i,1)=1;
        end
    end
end
```

Rest Tremor Amplitude

calculate velocity from acceleration

```
Hand_Vx=zeros(data_length,1);
Hand_Vy=zeros(data_length,1);
Hand_Vz=zeros(data_length,1);
```

```
for i=2:data_length
    Hand_Vx(i,1)=Hand_Vx(i-1,1)+Hand_Ax(i-1,1)/fs;
end
for i=2:length(Hand_Vy)
    Hand_Vy(i,1)=Hand_Vy(i-1,1)+Hand_Ay(i-1,1)/fs;
end
for i=2:length(Hand_Vz)
    Hand_Vz(i,1)=Hand_Vz(i-1,1)+Hand_Az(i-1,1)/fs;
end
testing_time=zeros(data_length,1);
for i=26:data_length-25
    if Hand_Vx<0.1&&Hand_Vz<0.1&&Hand_Vy<0.1
        EMG=EMG_hampel(i-25:1:i+25,1);
        EMG_change=max(EMG)-min(EMG);
        if EMG_change<20
            testing_time(i,1)=1;
        end
    end
end
```

Calculate Tremor Amplitude

```
raw_data_tremor_amplitude=Hand_Pr(testing_time);
R_hand=5;   % cm
Amplitude_upper=abs(max(raw_data_tremor_amplitude)-
mean(raw_data_tremor_amplitude));
Amplitude_lower=abs(min(raw_data_tremor_amplitude)-
mean(raw_data_tremor_amplitude));
Amplitude=2*R_hand*tan(max(Amplitude_upper,Amplitude_lower));
if Amplitude<=0.1
    disp('0: Normal')
elseif Amplitude<=1
    disp('1: Slight')
elseif Amplitude<=3
    disp('2: Mild')
elseif Amplitude<=10
    disp('3: Moderate')
elseif Amplitude>10
    disp('4: Severe')
else
    disp('Error')
end
```

Constance of Rest Tremor

```
expected_sample_num=100;    % cut the data into at least how many
 pieces
sample_size=floor(data_length/expected_sample_num);
sample_num=floor(data_length/sample_size);
tremor_amp=3;   % cm
tremor_count=0;
for i=1:sample_num
    test_data=data((i-1)*sample_size+1:i*sample_size,:);
```

```
    test_data_fft=fft(test_data);
    mag=abs(test_data_fft);
    df=fs/sample_size;
    freq=0:df:fs-df;
    mag_tremor=mag(freq>3&freq<7,:);
    mag_tremor_max=max(mag_tremor);
    for j=1:channel_num
        if mag_tremor_max(j)>tremor_amp
            tremor_count=tremor_count+1;
        end
    end
end
tremor_time=tremor_count/(sample_num*channel_num);
if tremor_time==0
    disp('0: Normal')
elseif tremor_time<=0.25
    disp('1: Slight')
elseif tremor_time<=0.5
    disp('2: Mild')
elseif tremor_time<=0.75
    disp('3: Moderate')
elseif tremor_time<=1
    disp('4: Severe')
else
    disp('Error')
end
```

Published with MATLAB® R2018a

MMatLab Code – Gravity Filter

```
data=load('raw.txt');
data1=load('pos.txt');
g=-9.81;
hax=data(:,3);
hay=data(:,4);
haz=data(:,5);
tax=data(:,12);
tay=data(:,13);
taz=data(:,14);
pax=data(:,21);
pay=data(:,22);
paz=data(:,23);
rax=data(:,30);
ray=data(:,31);
raz=data(:,32);
hqr=data1(:,1);
hqi=data1(:,2);
hqj=data1(:,3);
hqk=data1(:,4);
tqr=data1(:,5);
tqi=data1(:,6);
tqj=data1(:,7);
tqk=data1(:,8);
pqr=data1(:,9);
pqi=data1(:,10);
pqj=data1(:,11);
pqk=data1(:,12);
rqr=data1(:,13);
rqi=data1(:,14);
rqj=data1(:,15);
rqk=data1(:,16);
hax_new=hax+(hqi.*hqk+hqj.*hqr).*2*g;
hay_new=hay+(hqj.*hqk-hqi.*hqr).*2*g;
haz_new=haz+(hqr.^2-hqi.^2-hqj.^2+hqk.^2).*g;
tax_new=tax+(tqi.*tqk+tqj.*tqr).*2*g;
tay_new=tay+(tqj.*tqk-tqi.*tqr).*2*g;
taz_new=taz+(tqr.^2-tqi.^2-tqj.^2+tqk.^2).*g;
pax_new=pax+(pqi.*pqk+pqj.*pqr).*2*g;
pay_new=pay+(pqj.*pqk-pqi.*pqr).*2*g;
paz_new=paz+(pqr.^2-pqi.^2-pqj.^2+pqk.^2).*g;
rax_new=rax+(rqi.*rqk+rqj.*rqr).*2*g;
ray_new=ray+(rqj.*rqk-rqi.*rqr).*2*g;
raz_new=raz+(rqr.^2-rqi.^2-rqj.^2+rqk.^2).*g;
new_data=[hax_new,hay_new,haz_new,tax_new,tay_new,taz_new,...
    pax_new,pay_new,paz_new,rax_new,ray_new,raz_new];
dlmwrite('gravity_result.txt',new_data,'delimiter','\t')
```

Published with MATLAB® R2018a

MatLab Code – Low-pass and Bandpass Filter

Lowpass Filter

```
Fs = 100;
filtertype = 'FIR';
Fpass = 3;
Fstop = 4;
Rp = 0.1;
Astop = 80;
FIRLPF = dsp.LowpassFilter('SampleRate',Fs,...
                           'FilterType',filtertype,...
                           'PassbandFrequency',Fpass,...
                           'StopbandFrequency',Fstop,...
                           'PassbandRipple',Rp,...
                           'StopbandAttenuation',Astop);
NUM_LP = tf(FIRLPF);
dlmwrite('lowpass_coef.txt',NUM_LP,'delimiter','\t')
```

Bandpass Filter

```
Fs = 100;  % Sampling Frequency

Fstop1 = 2;                % First Stopband Frequency
Fpass1 = 3;                % First Passband Frequency
Fpass2 = 7;                % Second Passband Frequency
Fstop2 = 8;                % Second Stopband Frequency
Dstop1 = 0.0001;           % First Stopband Attenuation
Dpass  = 0.00057564620966; % Passband Ripple
Dstop2 = 0.0001;           % Second Stopband Attenuation
dens   = 20;               % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fstop1 Fpass1 Fpass2 Fstop2]/(Fs/2), [0
 1 ...
                          0], [Dstop1 Dpass Dstop2]);

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, Fo, Ao, W, {dens});
dlmwrite('bandpass_coef.txt',b,'delimiter','\t')
```

Published with MATLAB® R2018a